The DASH Network Communication Architecture

David P. Anderson
Robert Wahbe

November 8, 1988

ABSTRACT

DASH is an experimental distributed system intended for large high-performance networks connecting heterogeneous and mutually distrustful hosts. The DASH network communication architecture is based on the *Parameterized Message Channel* abstraction. It also includes a *subtransport layer* that incorporates many host-to-host functions, and a *remote operation facility* that provides request/reply communication. At higher levels, a *service access mechanism* and a *global naming system* together provide uniform global access to logical resources.

This report describes and discusses the DASH network communication architecture, and the DASH kernel implementation of this architecture.

Sponsored by the California MICRO program, AT&T Bell Laboratories, Digital Equipment Corporation, IBM Corporation, Olivetti S.p.A, and the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871. Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **08 NOV 1988**	2. REPORT TYPE	3. DATES COVERED **00-00-1988 to 00-00-1988**

4. TITLE AND SUBTITLE **The DASH Network Communication Architecture**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

DASH is an experimental distributed system intended for large high-performance networks connecting heterogeneous and mutually distrustful hosts. The DASH network communication architecture is based on the Parameterized Message Channel abstraction. It also includes a subtransport layer that incorporates many host-to-host functions, and a remote operation facility that provides request/reply communication. At higher levels, a service access mechanism and a global naming system together provide uniform global access to logical resources. This report describes and discusses the DASH network communication architecture, and the DASH kernel implementation of this architecture.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**60**	

1. INTRODUCTION

DASH is an experimental distributed system intended for large, high-performance networks; an overview is given in [3]. DASH includes a) an architecture for network communication, and b) a portable operating system kernel that implements this architecture. This report describes the network communication architecture and its implementation in the DASH kernel. [24, 25] describe other aspects of the DASH kernel.

The design of DASH is guided by the likely hardware advances of the next 5 to 10 years:

- Wide-area networks with low delay (30 milliseconds coast-to-coast) and high bandwidth (1 gigabit/second).
- Shared-memory multiprocessor workstations with 10 to 100 MIPS processing capacity.
- Large memory sizes, ranging from gigabyte workstation main memories to terabyte mass storage devices.

These advances will create the possibility of a new type of distributed system we call a *very large distributed system* (VLDS), spanning a large number (thousands or millions) of hosts, owned by many organizations and by individuals. A VLDS will support a range of new applications, such as:

- *Very large scale distributed parallel computation*: the set of processors on a VLDS will provide a ''processor bank'' numbering perhaps in the millions, and capable of supporting parallel computation at many levels of granularity.
- *Very large scale communication applications*: a VLDS will allow a variety of communication applications to be integrated into a single system: commercial applications (advertising, sales, and remote banking), interpersonal communication (mail, telephone, facsimile, and video conferencing), and distribution of digital audio and video entertainment and news.
- *High-bandwidth interfaces to distant resources*: many network services will offer graphics/audio-based interfaces. Such services will use data pipelining and caching to achieve the needed performance in the presence of inherently high network delays.

The communication architecture of DASH is designed to provide the basic facilities needed for these types of applications. The architecture spans multiple levels, from the network level up to naming and service access. The architecture uses the abstraction of a *Parameterized Message Channel* (or simply *channel*). A channel is a simplex message stream with several performance, reliability, and security parameters. The interface to the network-dependent communication layer is based on channels, and the channel abstraction appears at higher levels of the DASH system as well. Channels are the basis for a request/reply communication facility called the *Remote Operation Facility* (ROF).

The report is organized as follows: Section 2 is an overview of the DASH communication architecture. Section 3 presents the parameterized message channel abstraction. Section 4 lists the local kernel-level object types underlying the network communication abstractions, and Section 5 describes the mechanism for remote references to these objects. Sections 6 and 7 describe the network and subtransport levels. Sections 8 through 10 describe the Remote Operation Facility, the Service Access Mechanism, and the global naming system. Appendix I describes the *DASH Message Language* (DML), used for specifying network messages.

2. ARCHITECTURE OVERVIEW

We begin by giving an overview of the DASH network communication architecture. The structure of this architecture is shown in Figure 2.1.

2.1. The Network Layer

The DASH communication architecture can be implemented on multiple *networks*. Each network to which a DASH host is connected is represented in its kernel by a software module with a prescribed channel-based interface. These *network objects* provide host-to-host *network channels*. They encapsulate network-specific protocols for channel creation, deletion, and transmission, and other tasks such as routing and network management. The DASH *network layer* is the collection of these networks and the network objects.

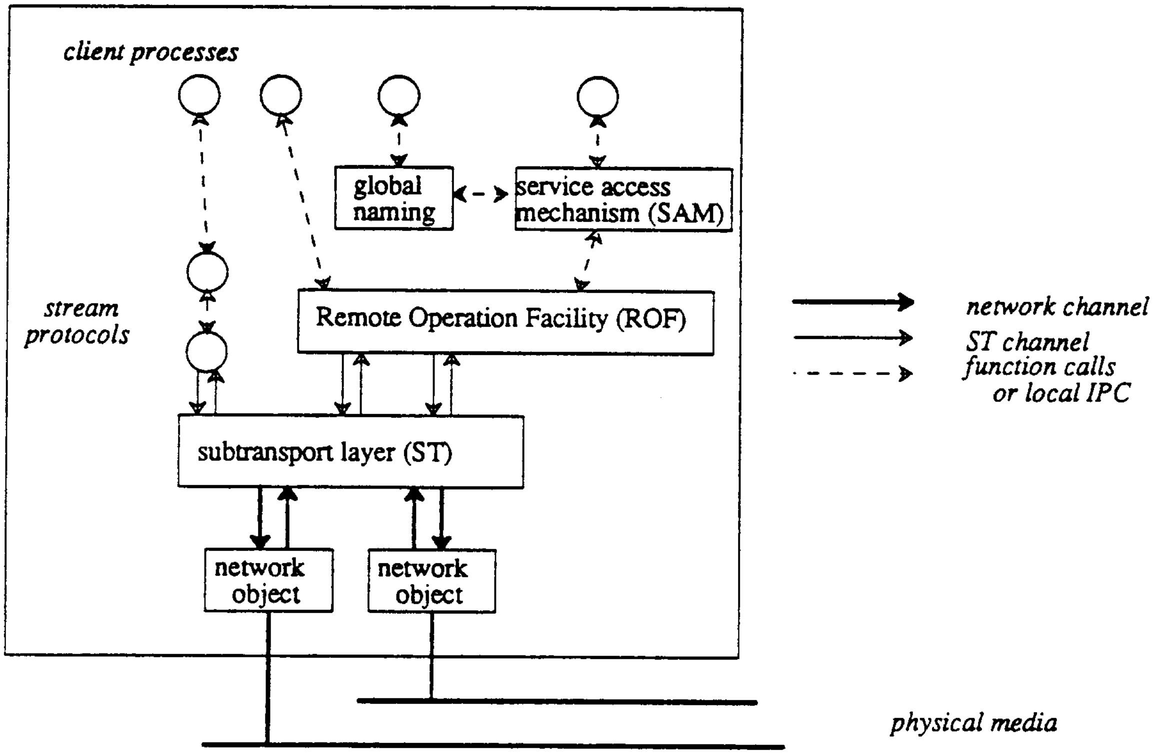

Figure 2.1: The DASH Communication Architecture.

2.2. The Subtransport Layer

The *subtransport layer* (ST) provides the basic form of inter-host process-level communication. All higher-level network communication in DASH passes through ST, and ST is the only direct client of the network layer. ST supports *ST channels*, which are ''value-added'' versions of network channels. ST provides communication security, does fragmentation and reassembly, multiplexes ST channels onto network channels, and arranges for ''fast acknowledgement'' of messages sent on ST channels.

In most existing protocol architectures, the functions of ST are done at higher levels (e.g., security is provided at the transport level). The DASH architecture has the advantage that these functions are consolidated into a single per host module. For example, we argue in [2] that a single secure channel between hosts is sufficient for authentication and privacy, rather than one per operation, session, or network.

2.3. Transport Protocols

The *Remote Operation Facility* (ROF) supports request/reply communication. It uses a set of ST channels called the *ROF connection*. Processes can also directly establish ST channels for stream-mode communication. The design of channel-based stream transport protocols is an area of future study.

2.4. Service Access

Certain remotely-accessible logical resources in a DASH system are called *services*. They can be accessed by clients through the *service access mechanism* (SAM). SAM provides *replication transparency*, *location transparency*, and *failure transparency*. It allows clients to use temporary capabilities (''service tokens'') to reduce name-resolution overhead.

2.5. Global Naming

A goal of the DASH communication architecture is that resources should be accessible from any host in the system. To satisfy this goal, a *global naming system* is used to name long-lived entities such as hosts, services, and owners. The system uses a tree-structured name space. Names are location independent in that 1) they do not imply the location of the named entity, and 2) they are the same regardless of the location or identity of the entity using the name.

The DASH global naming system and SAM are integrated. The global naming system is implemented as a set of services, and SAM uses the naming system for service location and authentication.

3. PARAMETERIZED MESSAGE CHANNELS

The DASH distributed system is intended to run on multiple types of computer architectures and communication networks. A large part of the DASH network communication system is *network independent*, and is based on a *network-dependent* part that has a network-independent interface.

In most existing distributed systems, the interface to the network level typically provides a simple abstraction such as unreliable, insecure datagrams. Upper layers then use this facility to provide higher-level abstractions such as reliable request/reply message-passing [7], reliable and secure typed message streams [20], or reliable byte streams [17]. Because the bottom-level abstraction (e.g., datagrams) is simple, it is easy to port the system to different network types. However, this approach suffers from several basic problems, stemming from the simplicity of the abstraction:

- Communication clients cannot express their performance, reliability and security needs to the communication provider. This results in wasted work. For example, data integrity is often a mandatory part of communication primitives, and is provided by software checksumming. This work is wasted for applications that do not require data integrity. Conversely, network interfaces may do data checksumming in hardware, but if this is concealed from upper layers, then checksumming may be redundantly done in software.
- Simple abstractions do not allow the communication provider to impose static limits on client behavior, such as the amount of client data outstanding within the network. The problem of congestion must then be attacked by methods (such as ICMP's source-quench messages [18]) that are often ineffective.
- No provisions are made for real-time performance guarantees. Such guarantees are needed for interactive high-bandwidth traffic such as digitized audio [4] and video.

In an attempt to solve these problems, the DASH network communication system is based on an abstraction called a *parameterized message channel*, or simply a *channel*. This decision is motivated by the projections summarized in Section 1. Current networks and protocol architectures do not directly support channels. However, our approach is capable of exploiting future advances in communication technology.

A channel links a *sender* to a *receiver*. It carries *messages*, which are untyped byte arrays. The sender invokes *send* operations on the channel. The receiver is typically a passive object such as a port; a message is considered *delivered* when it is queued on the port or given to a process waiting at the port. The sender and receiver are *channel clients*. The hardware and software system supporting the creation and use of channels is the *channel provider*. A channel client of one level may be an channel provider to a higher level.

A channel can be *deleted* by either client. A channel *fails* when one of the clients' hosts crashes, or when a failure or resource scarcity in the network makes it impossible to continue sending messages on the channel. When a failure occurs no further messages are delivered, and the surviving channel clients are notified of the failure. A channel is said to be *closed* when either it fails or it is deleted.

3.1. Channel Parameters

A channel has the following security and performance parameters. Some parameters represent guarantees by the channel provider; others are restrictions on the channel client. The form and meaning of some of the channel parameters depends on the *type* of the channel (see Section 3.2).

(1) **Authentication**: if true[1], then impersonation (delivery of a message not sent by the

[1] The authentication and privacy parameters are Boolean. They could instead be continuous, perhaps representing the strength of the underlying encryption system.

sender) is impossible.[2] This implies that the bit error rate (see below) is unaffected by the presence of malicious or malfunctioning hosts.

(2) **Privacy**: if true, then eavesdropping (access to a message by a host or process other than the receiver) is impossible.

(3) **Sequenced**: if true, messages are never delivered out of order.

(4) **Capacity**: an upper bound on the number of bytes of data outstanding (i.e., sent but not yet delivered) within the channel at any time. The clients are responsible for enforcing the channel capacity. If they fail to do so, the provider's guarantees are voided; messages may be delivered beyond the channel delay bounds or discarded.

(5) **Maximum message size**: an upper bound on the size of individual messages. This limit cannot be greater than the channel capacity.

(6) **Delay parameters**[3]: message delay is the elapsed real time between the start of the send operation and the moment of delivery. The components of this delay may include network transmission delay, queueing and processing delays at the sender and at intermediate switches, and processing at the receiver. Depending on the channel type, several parameters might be used. For example, a deterministic guarantee might use two parameters A and B, representing a delay bound of

$$A + B*(message\ size).$$

(7) **Workload parameters**: some channel types require parameters (such as average load and burstiness) describing the workload.

(8) **Message loss rate parameters**: a set of parameters describing the probability that a given message is successfully delivered. Messages may fail to be delivered because of 1) buffer overrun in the receiver host or in network switches, and 2) discarding due to checksumming. The form and meaning of the parameters depends on the channel type. The loss rate may be constant or load-dependent, and it may or may not depend on the message size. The loss rate does not take into account malicious or malfunctioning hosts. The channel abstraction does not specify freedom from denial of service.

(9) **Average bit error rate**: of the messages that are delivered, the fraction of bits that are correct. This parameter reflects the combination of 1) the error rate of the underlying transmission medium, and 2) the effectiveness of the checksumming algorithm. It is guaranteed by the channel provider.

(10) **Failure reporting delay bound**: this parameter is an upper bound on the interval between when the channel fails (see below) and when the surviving clients are

[2] "Impossible" assuming an unbreakable encryption scheme.

[3] Alternatively, a channel could have a "guaranteed bandwidth" parameter. However, this parameter is less convenient from the point of view of implementation, and it is determined by the current parameters as follows. If M is the maximum message size, D is the maximum delay of a message of size M, and C is the channel capacity, then a client can send a message of size M every DM/C seconds without violating the capacity rule, since at any point at most C/M messages (of total size C) will have been sent within the previous D seconds, and all earlier messages are guaranteed to have been delivered already. This will provide a bandwidth of about C/D bytes per second. The actual maximum bandwidth may either be lower (because of errors and protocol overhead) or higher (if actual delays are smaller than the upper bound).

notified of the failure.

A set of channel parameters is described by the following structure:

```
enum CHANNEL_TYPE {
      DETERMINISTIC,
      STATISTICAL,
      BEST_EFFORT
};

struct CHANNEL_PARAMS {
      CHANNEL_TYPE        type;
      BOOLEAN             authenticated;
      BOOLEAN             private;
      BOOLEAN             sequenced;
      U32                 capacity;
      U32                 max_msg_size;
      DELAY_PARAMS        delay_params;
      WORKLOAD_PARAMS     workload_params;
      LOSS_RATE_PARAMS    loss_rate_params;
      U32                 bit_error_rate;
      U32                 failure_delay_bound;
};
```

The meaning and form of `delay_params`, `workload_params`, and `loss_rate_params` is type dependent (see blow). The channel client is responsible for obeying the channel capacity and workload parameters; the channel provider is not responsible for detecting violations. In the event of a violation, the channel delay and message-loss parameters are voided, but the other parameters remain valid.

3.2. Channel Types

As indicated above, there are different *channel types*. A channel type consists of

- Parameterizations of the delay distribution, the workload, and the message loss rate.
- A partial order $<$ on the space of possible parameter values. For parameter values X and Y, $X < Y$ iff Y "dominates" X, i.e., Y is acceptable to a client whenever X is.

The following are examples of possible channel types:

Deterministic: the delay bounds are "hard"; only a channel failure will cause them to be violated. System resources (buffer space, media bandwidth) are allocated to individual channels. The channel provider rejects a channel request if its worst-case demands cannot be met with free resources.

Statistical: the delay bounds are statistical, perhaps involving its mean and variance[4]. The workload parameters include average load and burstiness. A channel creation request is rejected if either its expected message delay or its expected bit error rate (which is affected by the possibility of buffer overruns) is unacceptably high.

Best-Effort: channel creation requests are never rejected, and there are no workload parameters. Delay bound parameters are used only to schedule resources based on message delivery deadlines (see Section 3.5).

[4] Appropriate parameterizations of the delay distribution and of the workload are currently being investigated.

3.3. Channel Creation and Ownership

The creator of a channel (which may be either the sender or the receiver) *owns* the channel. It is "billed" for the channel if such a notion exists. Either side may delete the channel.

A set A of *actual* channel parameters is said to be *compatible* with a set R of *requested* parameters if

(1) the Boolean security parameters of A include those of R;

(2) the capacity and maximum message size parameters of A no less than those of R, and

(3) the performance parameters of A dominate (in the sense defined by the channel type) the parameters of R.

A channel creation request includes *desired* and *acceptable* parameter sets. [5] A channel creation succeeds only if the actual parameters of the resulting channel are compatible with the request's *acceptable* parameters. Furthermore, the channel provider tries to match the *desired* parameters as closely as possible.

3.4. Parameterized Message Channel Examples

As an example of the use of channel parameters, consider the case of a DASH client (say a transport protocol serving a user program) that requires data privacy. The protocol requests an ST channel (see Section 3.3). The desired and acceptable parameter sets both have the `private` flag set. ST creates the new ST channel, routing it through a network channel. Depending on the parameters of the network channel, several different situations are possible:

(1) Privacy is provided by ST-level data encryption.

(2) If the network has link-level encryption hardware, ST learns this from the network channel parameters, and does no data encryption.

(3) The network is considered secure, so no data encryption need be done at any level.

In any case, the optimal mechanism is used for privacy. If a client does not require privacy, no mechanism is used (which is again optimal). Without the channel security parameters, this optimization would not be possible. A similar situation exists for data integrity: the optimal checksumming mechanism can be used based on the values of the relevant channel parameters.

The following examples show the uses of the channel capacity and performance parameters:

- Initial request and reply messages in a request/reply protocol [5] should be sent on channels with a low delay bound. The precise delay bound and the delay bound type are determined by application needs. The channel capacity required may be large, unless it is known that request or reply messages will be small and infrequent.
- Request/reply retransmissions can be sent on channels with higher delay bounds.

[5] The current design allows only one "acceptable" parameter set. This could be extended to allow multiple sets.

- A stream protocol for bulk data transfer should use a high capacity, high delay channel for data [10].
- Reliability acknowledgements (for both request/reply and stream communication) should use low capacity, high delay channels.
- Flow control acknowledgements should use a low delay, low capacity channel.
- Digitized voice should use a high capacity, low delay channel, perhaps with a statistical delay bound [22]. A high bit error rate may be acceptable.
- Communication involving a human user interface traffic (such as for network window systems [11]) can tolerate a moderate amount of delay because of human perceptual limitations. The channel from user to application carries mouse and keyboard events, and can have low capacity. The channel in the opposite direction carries graphic information, and generally requires higher capacity.

In all these cases, the provider's knowledge of client needs increases the likelihood that they can be accommodated. For example, if packet queueing in an internetwork gateway is done using channel-specified deadlines, then a low-delay packet can be sent before high-delay packets that would otherwise cause it to be delivered late. A network may be capable of providing low delay or high capacity, but not both.

3.5. Process and Interface Scheduling

When an upper-level channel is created, its total delay is divided among its various stages (send protocol processing, ST channel delay, network channel delay, and receive protocol processing). When a message is sent on a channel, there is a *deadline* by which it must be handled (i.e., processed by a protocol, sent on a lower-level channel, or transmitted on a network medium). This deadline is the current real time plus the delay allocated to the next stage of the channel.

For channels whose delay includes processing time, these deadlines are used by the kernel's process scheduler to determine the execution order of protocol or user processes. For network channels, the deadlines are used to determine the order in which packets are queued for transmission on a network interface.

3.6. Issues in the Subtransport Layer

This section discusses issues that arise in building high-level channels on top of low-level channels. This is most directly relevant to the DASH subtransport layer.

3.6.1. Channel Caching and Multiplexing

ST *caches* network channels; i.e., it may retain a network channel even while it is not being used by any ST channel. This caching is motivated by two assumptions: 1) during a given time period a host will tend to communicate repeatedly with a small set of remote hosts; 2) it is slow and costly to create network channels.

ST does *upwards multiplexing* of multiple ST channels onto a single network channel (see Figure 3.2) [6]. This multiplexing can reduce overhead and delay compared to a

[6] It would also be possible to downwards-multiplex an ST channel across several network channels. If there were multiple networks paths between the hosts, this technique could be used to increase capacity

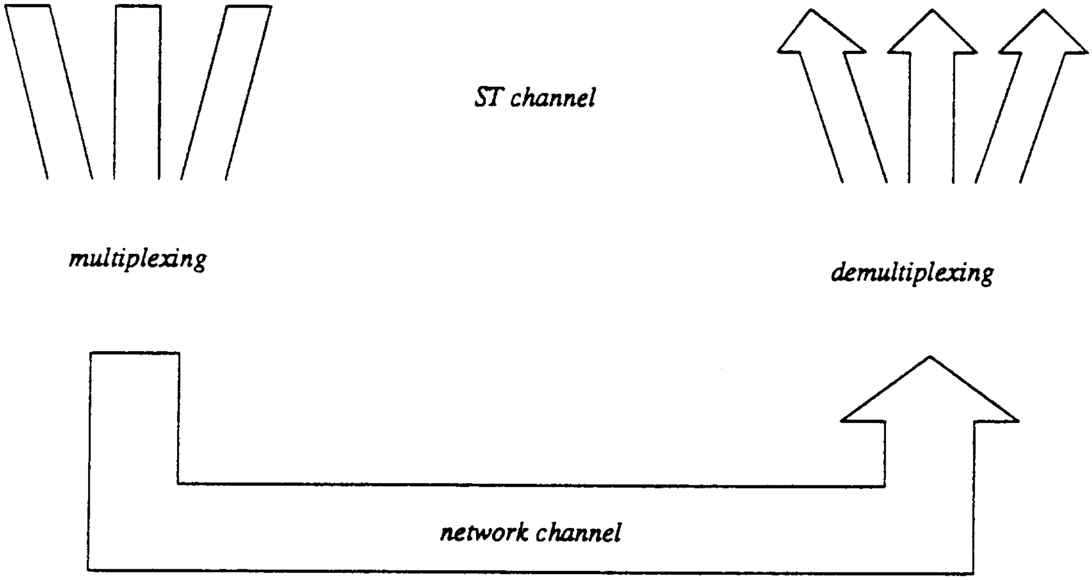

Figure 3.2: Channel Multiplexing

policy of creating a network channel for each ST channel.

The type and parameters of an ST channel may be different than those of the network channel on which it is multiplexed. In some cases ST can offer ''better'' parameters than those of the network channel. However, some parameters cannot be improved; multiplexing is subject to the follow restrictions:

- A deterministic ST channel can only be multiplexed onto a deterministic network channel, and a statistical ST channel can be multiplexed only onto a deterministic or statistical network channel.
- The delay bound parameters of an ST channel must be at least those of its network channel.
- The capacity of a network channel must be at least the sum of the capacities of its ST channels.

Algorithms for multiplexing decisions, in particular those involving statistical types and mixed types, remain to be studied.

3.6.2. Increased Maximum Message Size

At the network level, a message size limit is imposed by hardware restrictions, channel capacity, nonzero bit error rate, or the need for bounded delay and fairness. For example,

beyond that available in a single network channel. However, this has not been included in the DASH design because the expected gain may not outweigh the additional ST protocol complexity.

the Ethernet has a 1.5KB packet size limit [14]; future networks may have a limit of 64KB or so. It is an issue whether ST should offer larger maximum message sizes to its clients than those provided by the network layer. The following choices exist:

- The maximum message size of an ST channel is that of its network channel.
- The ST channel has a much larger size than its network channel (e.g., many megabytes). ST must incorporate a retransmission-based reliability mechanism; otherwise the message loss rate would increase exponentially with the ST message size. Because messages can exceed the capacity of the network channel, a flow-control mechanism also must be used. This general approach is taken in VMTP [8].
- The maximum message size of the ST channel may be larger than that of its network channel, but no greater than the capacity of the network channel, and not so large that a reliability mechanism is needed. This might be one or two orders of magnitude more than the network channel maximum message size (e.g., 64KB on an Ethernet).

ST uses the third option. This choice has the advantages of reducing the number of high-level messages (and hence protocol processing and scheduling overhead) without significantly adding to the complexity of ST, or requiring that it assume the role of a transport protocol.

ST chooses a maximum message size with the goal of maximizing potential throughput based on the combination of network channel error rate bound, ST channel error rate bound, and context switch time. ST does fragmentation and reassembly to support this larger message size. It does not retransmit fragments; if a message is incomplete when a fragment of the next message arrives, the partial message is discarded.

3.6.3. Failure Detection and Handling

If an ST channel has a smaller failure reporting delay bound than its network channel, ST must use ''pinging'' messages on the network channel to learn of host, process or network failures within the specified bound.

The handling of a network channel failure may vary. If the ST channel is best-effort, it may be possible to establish a new network channel (or switch to another existing network channel) and continue the ST channel service without interruption. For other ST channel types, the delay in establishing a new network channel might make this impossible.

3.6.4. Security and Reliability

If the ST channel has stricter security or reliability requirements than the network channel, techniques such as encryption and checksumming may be used to bridge the gap. These are discussed in Section 7.2.1.

If the ST channel is sequenced and the network channel is not, ST can include sequence numbers with messages and, depending on the other channel parameters, either reorder messages before delivery or simply discard messages that arrive out of order.

3.7. Flow Control and Channel Capacity Enforcement

At points where there can be speed mismatches, a communication system may do *buffering* to avoid message loss. *Flow control* can be used to avoid performance loss due to buffer overrun and dropped packets. Flow control mechanisms are often necessary even for minimal performance levels. On the other hand, flow control mechanisms may not be needed in certain situations (for example, if the speed of the sender is sufficiently low) and in that case may impose an unnecessary overhead.

The endpoints of communication are called the data *source* and *sink*. The source of data may be an I/O device (such as a disk, a main memory cache, or a real-time audio/video digitizer) or a computation (e.g., a process generating text or graphic images). It is useful to factor communication system buffers into three groups (see Figure 3.3):

(1) Buffers between the data source and the send side of the transport protocol.

(2) Buffers in the sender's kernel, the network switches and gateways, and in the receiver's network interface and low-level driver.

(3) Buffers between the receive side of the transport protocol and the data sink.

In a system based on parameterized message channels, these buffer groups can be treated separately. Cases in which no flow control is needed for one or more of the buffer groups can be identified, and a better transport protocol (simpler, faster or fewer messages) may be possible. In contrast, existing systems generally require end-to-end flow control [19].

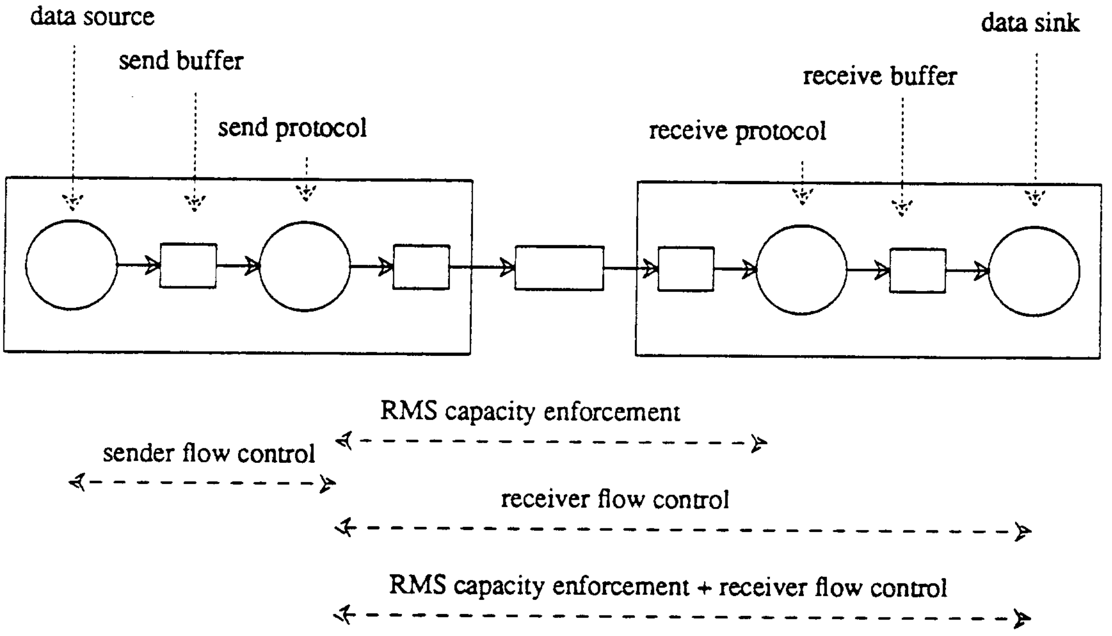

Figure 3.3: Types of Flow Control.

3.7.1. Channel Capacity Enforcement

The capacity parameter of a channel prevents overrunning buffers in group 2. In contrast, the flow control of TCP does not protect gateway buffers; ICMP source quench messages [15, 18] provide an *ad hoc* and often ineffective solution to this flow control problem.

The channel approach assumes that these buffers do not shrink spontaneously. If they do, it may be necessary for the channel provider to delete the channel, and for the clients to create a new channel.

As described in Section 3.1, capacity enforcement is the responsibility of the channel client; there are no capacity-enforcement mechanisms in the DASH ST or network layers. Depending on the channel parameters and the source and sink speeds, capability enforcement may not be needed. If it is needed, the following approaches are possible:

- *Rate-based*: using timers, the sender ensures that during any time period of duration $A + CB$, the number of bytes sent does not exceed C. This approach is pessimistic in the sense that it assumes the maximum delay for all messages.
- *Acknowledgement-based*: the sender receives *flow control acknowledgements* for messages received. This may achieve higher maximum throughput at the cost of the reverse message traffic.

These mechanisms, if needed, can be implemented by transport protocols. The same mechanisms may possibly be used to provide receiver flow control (see below).

3.7.2. Receiver Flow Control

If the receive transport protocol and data sink can process incoming data at a faster average rate than its arrival, it is possible that no flow control for buffer group 3 is needed [7]. If this condition does not hold, the protocol must stop sending data when the limit of the receive buffer is reached. A *receiver flow control* mechanism such as a sliding window protocol [17] can be used. The need for this flow control mechanism is independent of the need for channel capacity enforcement; if both are needed they could be integrated into a single protocol.

3.7.3. Sender Flow Control

If the data source can produce data faster than it can be sent on the ST channel (because of capacity enforcement, receiver flow control, or both) then there must be a *sender flow control* mechanism to protect buffer group 1. This is done in the DASH kernel using a flow controlled local IPC port for message-passing between the sender and the send protocol [24]. A sender blocks when a port queue size limit is reached. The sending transport protocol stops reading messages from the port while it is prevented from sending because of channel capacity enforcement or receiver flow control.

4. KERNEL STRUCTURE FOR NETWORK COMMUNICATION

Several abstractions (e.g. messages and message-passing objects) are used throughout the DASH network communication architecture. The objects which represent these

[7] A large receive buffer may be needed; the size is determined by several factors, including the variability of the receiver's speed.

abstractions are local to a host, and might be implemented in different ways on different hosts. In defining the network architecture, it is convenient to use language-level specifications in which these abstractions are represented by specific procedural interfaces.

In this section, we briefly describe the procedural interfaces that are used in the prototype DASH kernel to represent these abstractions. The DASH kernel is being written in C++, an object oriented language [21]. Therefore the interfaces are presented as class definitions. However, the architecture does not rely on an object oriented implementation, nor on the precise details of the interfaces.

4.1. Messages and Message-Passing Objects

Local communication is assumed to follow a message-passing paradigm (the DASH local message-passing system is described in [24]).

- A `MESSAGE` object represents data sent or received, either locally or remotely. The abstraction is an array of bytes, together with a header which includes an integer `type` field and a `flags` bitmask (for MP options).
- A `STREAM_MPO` object is a *message-passing object* (MPO) with unidirectional asynchronous semantics. Its basic operations are `send()` and `receive()`.
- A `REQ_REPLY_MPO` object is a message-passing object for synchronous (request/reply) operations. Its operations are `request_reply()` for the client, and `get_request()` and `send_reply()` for the server.

4.2. NAMED_ENTITY Objects

Entities with global names (see Section 10) are represented by objects of the base class `NAMED_ENTITY`. The following classes are derived from `NAMED_ENTITY`:

- An `OWNER` object represents an owner in the DASH global name space (see Section 10). Its data include the owner's public key (see Section 10) and, in some cases, the owner's private key.
- A `HOST` object represents a host, i.e., an endpoint of physical network communication. Its data include a list of the hosts network addresses, and the host's public key.

Other classes derived from `NAMED_ENTITY` will be introduced as needed.

5. REMOTE REFERENCES

The *remote reference* mechanism allows DASH hosts to refer to temporary entities (e.g., message-passing objects) in other hosts. A remote reference is created by a host to refer to an object within itself, and is issued to a *holder*, which may be either

- a particular host,
- all hosts on a particular network to which the host is connected, or
- all other hosts

A remote reference is a fixed-size (64-bit) datum; the issuing host determines the internal structure of the datum. Between crashes, a host must never issue the same remote reference twice (i.e., referring to two different objects) to a given holder.

The following remote reference values are reserved:

- 0 (NULL_REF) is the equivalent of a NULL pointer, i.e. no object is referenced.
- 1 (ROF_NOTIFY_MPO) refers to an MPO used to notify the ROF module of the creation or failure of ST channels.
- 2 (ST_NOTIFY_MPO) refers to an MPO used to notify the ST layer of the creation or failure of network channels. In the DASH kernel, the only client of the network layer is ST and ST_NOTIFY_MPO is not needed. However, by having the notification MPO specified, other network layer implementations, after bootstraping the connection using this well-known reference, may support multiple notification ports and clients.

5.1. Implementation of Remote References

A host must maintain the set of remote reference it has issued, the holders to which they were issued, and the type of the object to which the remote reference refers. The mechanism to authenticate the source of a message provided by ST can be used to insure that remote references are used only by those who have permission to do so. Hosts that allow objects to be deleted must also have a mechanism for detecting ''dangling'' remote references. This can be done using unique IDs' (see below).

In the DASH kernel, a remote reference is represented as a pair <*unique object identifier (UID), session identifier*> of 32-bit numbers. The UID is a sequence number assigned when a remotely-referenced object is created, and stored with the object. The session identifier is used to detect stale remote references.

Each entity to whom remote references are issued manages those entries using a REMOTE_REF_MGR object. The REMOTE_REF_MGR class provides the following interface:

```
struct REMOTE_REF {
      U32    object_uid;
      U32    session_id;
};

enum {
      STREAM_MPO_TYPE,
      OUT_NET_CHANNEL_TYPE,
      IN_NET_CHANNEL_TYPE,
      OUT_ST_CHANNEL_TYPE,
      IN_ST_CHANNEL_TYPE,
      HOST_TYPE,
      OWNER_TYPE,
      NO_TYPE
} OBJECT_TYPE;
```

```
BOOLEAN                                                  // TRUE implies a new entry
REMOTE_REF_MGR::add_entry (
        SHARED_OBJECT*      object_pointer,        // pointer to object
        OBJECT_TYPE         object_type,           // object type
        PTR_OR_INT          data                   // data associated with entry if new
        PTR_OR_INT*         existing_data          // returned if not a new entry
        REMOTE_REF*         remote_ref             // returned
);

SHARED_OBJECT*                                           // NULL if then entry does not exist
REMOTE_REF_MGR::get_entry (
        REMOTE_REF*         remote_ref,
        OBJECT_TYPE         object_type,
        PTR_OR_INT*         existing_data          // returned
);
```

Duplicate remote references in the table are not allowed; if an entry for the object already exists, it is returned. There are also functions to delete entries and change the data associated with an entry.

6. THE NETWORK LAYER

In DASH terminology, a *network* is an abstract entity that connects a set of hosts and provides parameterized message channel service between pairs of these hosts. Different networks need not be physically or logically disjoint. For example, the DARPA Internet and a local Ethernet (with the addition of support for channels) could be two separate DASH networks, although they might share host interfaces and network media.

Each DASH host is connected to one or more networks, and must implement the channel protocols of those networks. A host has one or more *network addresses* in each network to which it is connected. In the current design, there is no mechanism to connect channels of different networks. A pair of DASH hosts can communicate directly only if they belong to a common network.

In the DASH kernel, the network layer consists of a set of objects derived from the base class NETWORK. Each of these objects corresponds to a network to which the host is connected. A NETWORK object has the following Boolean attributes:

All hosts trusted: true if this kernel trusts the kernels of all the hosts on the network. This does not imply that the network is physically secure.

Physical broadcast network: true if the network has the property that any message completely received by any node is received by all nodes. LAN's such as Ethernet [14] have this property.

Network objects should support only those channel security and reliability properties for which it has hardware support (such as by hardware checksumming or encryption in the network interface) or that hold by virtue of trust properties (e.g., that the network is physically secure). The gap between these properties and those required by clients is bridged by ST.

6.1. Network Channel Endpoint Objects

A network channel is represented, in each of the kernels it connects, by an *endpoint object*. An endpoint is *active* on the host that created the channel, and *passive* on the other end. It is *outgoing* or *incoming* depending on the data direction relative to an endpoint. There are two network endpoint base classes, OUT_NET_CHANNEL and IN_NET_CHANNEL. For each type of network, there are network channel endpoint classes derived from these base classes. The classes are derived from a base class containing a *type* field that is used to distinguish between the endpoint types in operations such as NETWORK::delete() that take generic pointers.

6.2. Network Channel Creation: Active End

The NETWORK class provides the following virtual function to create a set of network channels:

```
NET_CHANNEL_CREATE_STATUS
NETWORK::create_channel (
      HOST*                 destination,         // destination host
      NETWORK_ADDRESS       network_address,     // network address to use
      U32                   to_peer,             // opaque data delivered to peer
      U32*                  from_peer,           // opaque data returned from peer
      int                   desc_count,          // number of channels requested
      NET_CHANNEL_REQUEST[]    descs             // list of requests descriptors
);

struct NET_CHANNEL_REQUEST {
      CHANNEL_DIRECTION direction;               // CHANNEL_IN or CHANNEL_OUT
      CHANNEL_PARAMS*   desired;
      CHANNEL_PARAMS*   acceptable;
      STREAM_MPO*       data_mpo;                // for incoming channels only
      STREAM_MPO*       closure_mpo;             // closure notification MPO
      U32               closure_data;            // delivered in closure notification
      CHANNEL_ENDPOINT* endpoint;                // endpoint object (returned)
} NET_CHANNEL_REQUEST;

enum NET_CHANNEL_CREATE_STATUS {
      CREATE_ACCEPTED,                           // request succeeded
      CREATE_FAILED,                             // request failed
      CREATE_REJECTED                            // request rejected by peer ST
};
```

NETWORK::create_channel() attempts to establish a set of network channels. Either the entire set of channels is created or none are. Each channel in the request is described by a NET_CHANNEL_REQUEST structure. The actual parameters of the resulting channels are not returned, but are public data of the endpoint object.

6.3. Network Channel Creation: Passive End

ST is notified of network channel creation by request/reply *notification operations* directed from the network layer to the ST_NOTIFY_MPO. These operations inform the passive ST (the host receiving the request) of a network channel creation request. It may either accept or reject the request. The request message format is:

```
struct NET_NOTIFY_CREATE_REQUEST {
      HOST*                 source;           // active host
      U32                   opaque_data;      // from NETWORK::create_channel()
      U32                   desc_count;       // number of channels in request
      NET_CREATE_REQUEST_DESC[] descs;        // array of descs
};

struct NET_CREATE_REQUEST_DESC {
      CHANNEL_DIRECTION direction;            // CHANNEL_IN or CHANNEL_OUT
      CHANNEL_PARAMS    actual_params;        // parameters of this channel
      CHANNEL_ENDPOINT* endpoint;             // local channel endpoint
};
```

The reply message format is:

```
struct NET_NOTIFY_CREATE_REPLY {
      BOOLEAN               accepted;         // FALSE ==> rejected
      U32                   opaque_data;      // returned to peer ST
      NET_CREATE_REPLY_DESC[] descs;          // present only if accepted
};

struct NET_CREATE_REPLY_DESC {
      STREAM_MPO*           data_mpo;         // for incoming channels only
      U32                   closure_data;     // included in closure notification
      STREAM_MPO*           closure_mpo;      // closure notification MPO
};
```

6.4. Network Channel Deletion

The NETWORK class provides the following virtual function to delete network channels:

```
void
NETWORK::delete (
      U32                   to_peer,          // delivered to peer
      int                   endpoint_count,   // number of channels to be deleted
      CHANNEL_ENDPOINT[]          endpoints   // list of local net channel endpoint
)
```

This deletes the specified network channels and their associated endpoints on the local host. Network channels may be deleted by either client.

6.4.1. Closure Notification

ST is informed of channel closure (both failures and deletions by the peer) by messages delivered to the channel's closure MPO. Failure notifications are guaranteed to be delivered within the failure_delay_bound channel parameter. Network channel closure notification messages have their type field set to NET_CLOSE_TYPE, and have the following format:

```
struct NET_NOTIFY_CLOSE {
      NET_CLOSE_REASON  close_reason;         // reason channel was closed
      CHANNEL_ENDPOINT* local_endpoint;       // local network channel endpoint
      U32               my_opaque;            // data supplied upon channel creatic
      U32               peer_opaque;          // data supplied by peer
};
```

```
enum NET_CLOSURE_REASON {
      DELETED_BY_PEER,
      PEER_CRASH,
      NETWORK_FAILURE,
};
```

My_opaque is the closure_data supplied locally when the channel was created. If the channel was deleted by the peer, peer_opaque is the data supplied by the NETWORK::delete() operation of the peer. ST is responsible for eventually deleting the network channel endpoint objects after receiving the notification message; this prevents any ``dangling pointer'' problem.

6.5. Network Channel Data Messages

The OUT_NET_CHANNEL base class provides the following virtual function to send a message:

```
OUT_NET_CHANNEL::send (
      MESSAGE* message
);
```

The header of message contains a DEADLINE flag. If set, the header contains a deadline for message transmission; this should not precede the current real time plus the guaranteed delay of the channel. This allows the client to specify that a message has *less* stringent delay parameters than that normally associated with the channel. The deadline is used for queueing when the network interface has a nonempty transmission queue.

Each network channel has an associated *data MPO* to which data messages are delivered. Data messages have their type field set to NET_DATA_TYPE to distinguish them from closure notification messages.

6.6. Security Considerations

A single host may be connected to both secure and insecure networks. To insure that messages from an insecure network do not interfere with authenticated channels, each network maintains a table of valid MPO's and will deliver messages only to these.[8]

If a network is not physically secure, its channel establishment protocol will be insecure, and channel creation notification messages may have false remote host names. Such a ``spurious'' channel does not represent a security violation; it will be detected and rejected by ST.

7. THE SUBTRANSPORT LAYER

All network communication in DASH (except for network-internal tasks such as routing and network management) passes through the subtransport (ST) layer. ST enhances the channel service provided by the network layer in the following ways:

- ST multiplexes ST channels onto network channels and caches idle network channels.

[8] We currently assume that each network uses a sperate network interface. Having networks share interfaces is currently under investigation.

- ST uses encryption and/or checksumming mechanisms to improve security and error rate parameters, as needed.
- ST does fragmentation and reassembly so that the maximum message sizes available to its clients can exceed that offered by the network layer.
- ST provides "fast acknowledgements" that clients can use to provide flow control and capacity enforcement.

The clients of ST include stream transport protocols and the remote operation facility (ROF).

7.1. ST Client Interface

Clients interact with ST through a combination of procedure calls and message-passing operations. The procedural interface allows clients to initiate operations such as channel creation (for the remainder of this section "channel" refers to ST channels only, not network channels). The message-passing interface is used to inform clients of events such as channel creation and closure, and to allow them to send and receive messages on existing channels.

The interface for creating and closing channels involves the following objects (see Figure 7.1):

- The ST layer is represented by the `SUBTRANSPORT` module. Its operations include channel creation and deletion.

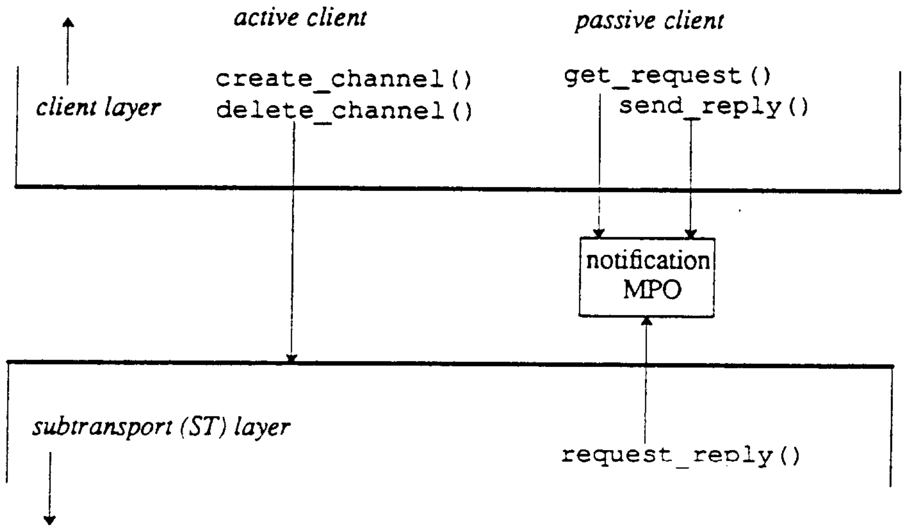

Figure 7.1: Client Interface for Creating and Deleting Channels.

- Notification of channel creation is done by performing request/reply operations on client-specified *notification MPO's*.

The interface for interactions with existing channels involves the following objects (see Figure 7.2):

- A channel endpoint is represented on its local host by an *endpoint object*. An ST endpoint is *active* on the host that created the channel, and *passive* on the other end. It is *outgoing* or *incoming* depending on the data direction relative to an endpoint. There are two ST endpoint base classes, `OUT_ST_CHANNEL` and `IN_ST_CHANNEL`. `OUT_ST_CHANNEL` is derived from the `STREAM_MPO` base class. Both endpoint classes are derived from a base class that includes a `type` field, so their types can be determined from generic pointers.
- Each channel has an associated stream-mode *data MPO* at its receiving end. Client messages are sent to this MPO by default.
- Each channel has an associated stream-mode *closure MPO* at each end. In the event of channel closure, a message is sent to this MPO.
- Each channel has an associated stream-mode *acknowledgement MPO* at its outgoing end. Fast acknowledgement messages are sent to this MPO.

The stream-mode messages sent by ST include a value in the `type` field of the message header. These values include

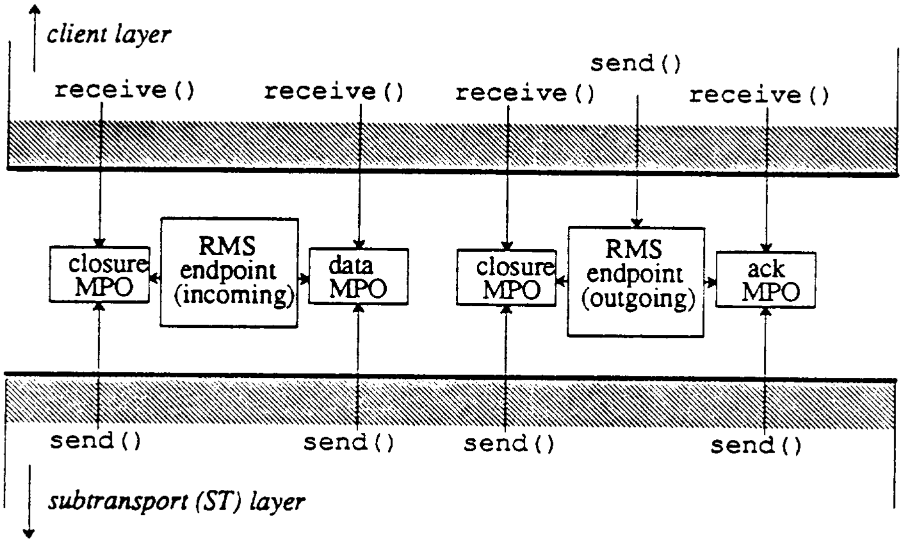

Figure 7.2: The Client Interface for Channel Usage.

ST_DATA_TYPE: the message is channel client data.

ST_REDIRECTED_DATA_TYPE: if a channel client data message is directed to a MPO other than the default data MPO, but the MPO remote reference is invalid, it is assigned this type and delivered to the default data MPO.

ST_CLOSURE_TYPE: the message is a notification of channel closure.

ST_ACK_TYPE: the message is a fast acknowledgement.

The presence of this type field allows clients to use a single MPO to handle multiple message types. For example, a single MPO may serve as both the data MPO and closure MPO for an incoming channel.

7.1.1. Channel Creation: Active End

The SUBTRANSPORT module provides the following operation for creating ST channels:

```
ST_CHANNEL_CREATE_STATUS
SUBTRANSPORT::create_channel(
        HOST*               destination,
        REMOTE_REF*         notify_mpo,           // who to notify at peer
        U32                 to_peer,              // passed to peer in notification
        U32*                from_peer,            // returned from peer
        int                 desc_count,           // number of channels to create
        ST_CREATE_DESC[]    descs                 // array of per-channel descriptors
);

struct ST_CREATE_DESC {
        CHANNEL_DIRECTION   direction;            // CHANNEL_IN or CHANNEL_OUT
        CHANNEL_PARAMS*     desired;
        CHANNEL_PARAMS*     acceptable;
        CHANNEL_ENDPOINT*   endpoint;             // channel endpoint object (returned)
        U32                 to_peer;              // passed to passive client
        U32                 from_peer;            // returned from peer
        STREAM_MPO*         data_mpo;             // incoming only
        STREAM_MPO*         closure_mpo;          // MPO for closure notification
        U32                 closure_data;         // opaque data in closure message
        STREAM_MPO*         ack_mpo;              // outgoing only
};

enum ST_CHANNEL_CREATE_STATUS {
        CREATE_ACCEPTED,                          // request succeeded
        CREATE_FAILED,                            // request failed
        CREATE_REJECTED                           // request rejected by peer
        INVALID_NOTIFY_MPO                        // invalid notify mpo was specified
};
```

SUBTRANSPORT::create_channel() is used to create a set of channels to the specified host. Either the entire set is created or none are. The peer is notified at notify_mpo. To_peer is passed to the passive client in the notification message; it can be used to identify the request. If the peer rejects the request, from_peer can be used to provide an error code. Each channel is described by an ST_CREATE_DESC structure. If the operation succeeds, a pointer to the channel endpoint object is returned in endpoint. The to_peer and from_peer fields in the ST_CREATE_DESC allow clients to exchange a word of data associated with each channel. The actual parameters of the channel are public data of the object. `Data_mpo`, for incoming channels, specifies the data MPO (for outgoing channels, the data MPO is specified by the passive ST client). Closure notification messages will be written to `closure_mpo`. For outgoing channels, `ack_mpo` specifies a stream MPO to which fast acknowledgements (see Section 7.1.6) are to be sent.

7.1.2. ST Creation: Passive End

ST notifies the passive client of attempts to create channels by doing a request/reply operation on a *notification MPO*. The passive client may either accept or reject the creation request. If it accepts, the reply message specifies data, acknowledgement, and closure MPO's for each channel. The passive client is also notified when an attempted ST channel creation request fails, e.g., because a network channel of sufficient capacity could not be created. Notification request messages have the following structure:

```
struct        ST_NOTIFY_REQUEST {
        HOST*                 source;           // active host
        BOOLEAN               success;          // was the creation successful?
        int                   desc_count;       // number of channels
        ST_NOTIFY_REQ_DESC[] descs;             // defined only if successful
};

struct        ST_NOTIFY_REQ_DESC {
        BOOLEAN               direction;        // true iff outgoing
        void*                 endpoint;         // local channel endpoint object
        U32                   opaque_data;      // from create_channel() call
};
```

Notification reply messages have the following form:

```
struct        ST_NOTIFY_REPLY {
        BOOLEAN                       accepted;
        ST_NOTIFY_REPL_DESC[]         descs;    // present only if accepted
};

struct        ST_NOTIFY_REPL_DESC {
        U32                   opaque;           // for closure message
        STREAM_MPO*           data_mpo;         // incoming only
        STREAM_MPO*           ack_mpo;          // outgoing only
        STREAM_MPO*           closure_mpo;      // where to deliver closure message
};
```

The data items returned by the passive client in its `ST_NOTIFY_REPLY` message are analogous to those supplied by the active client in the `SUBTRANSPORT::create_channel()` call above.

7.1.3. ST Channel Creation Scenarios

The first ST channels established to a given host are those created by the ROF module when it first invokes a remote operation on that host. The establishment of subsequent channels follows this scenario:

(1) The *active* client contacts the *passive* client via ROF. The passive client creates a notification MPO and obtains a remote reference for it, which is included in the ROF reply message.

(2) The active client receives the ROF reply message.

(3) The active client calls SUBTRANSPORT::create_channel(), and the two ST modules set up the requested channels.

(4) When the channels have been created, the passive ST does a request/reply operation on the notification MPO. If the channel group is rejected, the channels are deleted.

(5) The call to SUBTRANSPORT::create_channel() returns at the active end with the appropriate status information.

7.1.4. ST Channel Deletion

The SUBTRANSPORT module provides the following operation for deleting ST channels:

```
void
SUBTRANSPORT::delete_channel(
        U32*                opaque,             // array of opaque data
        int                 endpoint_count,     // number of channels to delete
        CHANNEL_ENDPOINT[] endpoints            // endpoint objects
)
```

This deletes the specified channels and the corresponding local endpoint objects. The peers may be on different remote hosts. For each channel, a *closure message* (of type ST_CLOSURE_TYPE) containing the corresponding element of opaque is delivered to the peer's closure MPO if possible. The client is responsible for the deleting the endpoint after receiving the closure notification message. Having the client delete the endpoint solves any ''dangling pointer'' problem. A closure message has the following structure:

```
struct      ST_CLOSURE {
        CHANNEL_ENDPOINT* endpoint;            // local endpoint object
        ST_CLOSURE_REASON reason;
        U32               my_opaque;           // supplied at channel creation
        U32               peer_opaque;         // defined only for PEER_CLOSE
};

enum    ST_CLOSURE_REASON {
        DELETED_BY_PEER,
        PEER_CRASH,
        NETWORK_FAILURE
};
```

7.1.5. ST Channel Data Messages

A data message can be sent on an outgoing channel using the STREAM_MPO::send() operation on its endpoint object. The message header contains a bitmask with the following flags:

AUTHENTICATE_SENDER

This is meaningful only on a host-authenticated ST channel. The message header contains a pointer to an OWNER object. On delivery, the incoming message will include a pointer to a corresponding OWNER object in the peer kernel. If the source kernel is *security correct* [1] (i.e., if it provides privacy and authentication locally) the receiver knows that the message originated from OWNER. If the source kernel is not security correct, it knows only that the source kernel possesses the private key of OWNER.

AUTHENTICATE_RECEIVER

This is meaningful only on a host-authenticated ST channel. The outgoing message header contains a pointer to an OWNER object. ST will verify that the owner is present on the destination host before sending the message.[9] However, there is no guarantee that the message is actually delivered to that owner.

DESTINATION_MPO

The message header contains a remote reference to an MPO on the remote host; the message is to be delivered to this MPO (instead of the default data MPO).

ACK_REQUESTED

The message header contains a 32-bit message ID; ST sends an acknowledgement message containing this ID to the acknowledgement MPO when this message is delivered.

DEADLINE

The message header contains a deadline for transmission. This deadline should not precede the current real time plus the guaranteed delay. This allows the client to specify that a message has less stringent delay parameters than those normally associated with the channel. This deadline determines the message's queueing priority at the network level.

If the DESTINATION_MPO flag is not set in a STREAM_MPO::send() operation, or if an invalid remote reference if given, the message will be delivered to the channel's default data MPO (in the latter case the message is assigned type ST_REDIRECTED_DATA_TYPE). This facility is used by ROF to handle retransmitted replies sent to MPO's that have been deleted.

The header of a message delivered by ST includes a copy of the flags bitmask supplied by the sender. If the AUTHENTICATE_SENDER flag was set, the header will include a pointer to an OWNER object representing the sender.

7.1.6. Fast Acknowledgements

If a message is sent with the ACK_REQUESTED flag set, ST will (unreliably) send an acknowledgement message to the channel's acknowledgement MPO after the original message has been delivered. The acknowledgement message has its type set to ST_ACK_TYPE, and has the following format:

```
struct      ST_NOTIFY_ACK_REPLY {
      U32   message_id;
};
```

This facility can be used for acknowledgement-based flow control. ST-level acknowledgements may be preferable to higher-level acknowledgements, since the acknowledgement can be sent earlier. The facility does not replace higher-level reliability acknowledgements.

[9] We have not yet defined precisely what it means for an owner to be present on a host.

7.2. The Subtransport Protocol

This section describes the *DASH Subtransport Protocol*. This protocol is used for communication between ST layers, and must be implemented on all DASH hosts. The protocol consists of two related subprotocols: The *ST Control Protocol* is used for secure channel establishment, owner certification, ST channel creation and deletion, fast acknowledgements, and pinging. The *ST Data Protocol* is used for conveying client messages.

7.2.1. Security Encapsulations

Recall from Section 3.1 that the parameters of a channel include the following:

- **Private**: true if eavesdropping is impossible.
- **Host-authenticated:** true if impersonation by another host is impossible.
- **Bit error rate**: the long-term average bit error rate.

In addition, a given network has the parameters

- **All Hosts Trusted**: this flag is true if this host trusts the kernels of all hosts on the network. This trust simplifies owner certification, but is orthogonal to network security; it does not imply that the network is private or host-authenticated.
- **Physical Broadcast Network**(PBN): this flag is true if any message received completely by any host on the network is received by its addressee.

The messages sent by ST have varying security and error rate requirements. Indeed, the requirements may vary between the different fields of a message. ST uses the following mechanisms:

- **Cleartext**: the message is sent verbatim, with no additional data.
- **Encryption**: part or all of the message is encrypted using DES single-key encryption [16]. A *channel key* shared by the two hosts is used. The 64-bit remainder of the encryption is appended to the cyphertext. This provides privacy, authentication and error detection.
- **Cryptographic Checksumming**: part or all of the message is sent in cleartext, but the 64-bit remainder from its DES encryption is appended. The provides a ''cryptographic checksum'' having the property that it is virtually impossible to modify the data without modifying the checksum, so this mechanism provides both error detection and authentication.
- **Checksumming**: part or all of the message is sent in cleartext, but is followed by a 32-bit checksum. This provides error detection. However, it does not provide authentication (even if the checksum were encrypted) because it is possible to modify the data in such a way that the checksum is unchanged.
- **Encrypted Trailer**: this technique is used only when communicating on a PBN. A 64-bit *trailer*, encrypted with the channel key, is appended to the entire message (not just to the part being authenticated). The trailer is a 32-bit sequence number followed by a 32-bit timestamp in seconds. The receiver decrypts the trailer, and accepts the message if it lies within an acceptable range (in terms of both sequence number and time) of the previous packet received. This provides authentication.

These properties are summarized in Table 7.1.

technique	*privacy*	*authentication*	*error detection*
cleartext	no	no	no
encryption	yes	yes	yes
checksum	no	no	yes
cryptographic checksum	no	yes	yes
encrypted trailer	no	yes	no

Table 7.1: Properties of Security Mechanisms.

The combination of mechanisms used for a particular message (or part of a message) is called its *security encapsulation.* The ST can use any combination of the above mechanisms to achieve the needed properties. The choice depends on what is provided by the network layer. The first four techniques are mutually exclusive. Encrypted trailers may be used alone or in combination with checksumming.

ST is free to use any security encapsulation that has the needed properties. The choice may depend on the architecture (processor and encryption/checksumming hardware). Assume that 1) in order of increasing cost, the techniques are

cleartext
encrypted trailer
checksum
cryptographic checksum
encryption

2) that the cost differences are all nonnegligible, and 3) that techniques can be intermixed within a message with no additional cost. With these assumptions, the most efficient security encapsulation can be determined as follows:

```
if (ST channel is private and network channel is not private) {
    use encryption;
} else {
    if (ST channel is authenticated and network channel is not authenticated)
        if (network is PBN) {
            if (ST channel error rate < network channel error rate) {
                use encrypted trailers and checksumming;
            } else {
                use encrypted trailers;
            }
        } else {
            use cryptographic checksumming;
        }
    } else {
        if (ST channel error rate < network channel error rate) {
            use checksumming;
        } else {
            use cleartext
        }

    }
}
```

7.2.2. Mixed Encapsulations

The security and reliability requirements for the header of ST data messages (see Section 7.2.6) may differ from those of the data part. Therefore different security encapsulations may be used for the two parts. For example, a message might have the form

ST header (cleartext)
cryptographic checksum of ST header
user data (cleartext)
checksum of user data

Checksums (regular or cryptographic) of the two parts are not merged. If both parts use an encrypted trailer, only one is appended to the message.

7.2.3. Encapsulation Type is Implicit

There is no need to encode the security encapsulation of each message within the message itself. The ST can deduce the encapsulation as follows.

- Control and data messages can be distinguished because they arrive on different network channels, and hence on different MPO's.
- The encapsulation of control messages is determined when the control connection is established.
- The header encapsulation of data messages is determined when the network channel is established. From the ST channel remote reference contained in the header (see Section 7.2.6), the ST can determine the ST channel for which the message is destined. The data encapsulation for an ST channel is determined when the channel is established.

7.2.4. Security Considerations

Recall from Section 6.6 that network channel data messages are inherently network-authenticated. There is therefore no problem in mixing secure (i.e., private and authenticated) and insecure networks in the same host. For example, suppose a host is connected to network *A*, which is secure, and network *B*, which is insecure. If a secure ST channel is established using a network channel on network *A*, then no encryption will be used. A malicious host on network *B* can indeed send forged messages, but they will not interfere with the secure ST channel.

7.2.5. The ST Control Protocol

The ST control protocol from host *A* to host *B* consists of a sequence of synchronous request/reply operations from *A* to *B*[10]. Operations from *B* to *A* may overlap these operations. A simple retransmission protocol is used for reliability. The control protocol also includes *fast acknowledgements*, which are unreliable unidirectional messages.

An *ST control connection* between two hosts consists of a pair of network channels between the hosts, one in each direction, each created by its sending end. All ST control messages are sent on these channels. The ST control protocol involves relatively small and infrequent messages. Hence the network channels can have a small capacity, but should have minimal delay.

A host *A* may initiate the creation of a control connection to a host *B* simply by creating the initial network channel to *B*. *B* may accept the connection by creating a network channel to *A*, or may reject it by rejecting the original channel creation request. No synchronization problem arises if two hosts simultaneously create a control connection to one another.

A *subtransport secure connection* exists between two ST modules *A* and *B* if:

- *A* and *B* have a means for sending private and host-authenticated messages. In some cases this can be done by using private or host-authenticated network channels. More generally, the ST modules must have agreed upon a common secret encryption key.
- *A* and *B* have a means for certifying to each other what owners (kernel clients) they represent. They have this means if 1) they have agreed upon on a pair of *owner certification strings* that can be used to prove the possession of owner private keys, or 2) they trust each other.

A *secure connection* is an extension of a control connection; it is established only after a control connection has been established, and ceases to exist if the control connection fails. Secure connection establishment is done by message exchange using Diffie-Hellman public-key encryption [2]. Once the secure connection has been established, DES single key encryption is used. The first operation on a control connection is secure connection establishment. This operation is always initiated by the host with the lexicographically greater name. To establish a secure connection to host *B*, host *A* generates a random channel secret key *S* and sends the following request message:

[10] If experiments prove this serialization to be a performance bottleneck, the protocol will be changed to allow parallel requests.

```
msg_typedef struct {
        SKE_KEY                     key;            // the connection secret key
        BYTES                       dest_name;      // name of destination host
        CERT_STRING                 cert_string;
        SECURITY_ENCAPSULATION      encapsulation;
        CHECKSUM                    checksum;
} ESTABLISH_SECURE_CHANNEL_REQUEST;

msg_typedef struct {
        enum {      CLEARTEXT,
                    CHECKSUM,
                    CRYPTO_CHECKSUM,
                    ENCRYPTED
        } data_mode;
        flags {encrypted_trailer};
} SECURITY_ENCAPSULATION;
```

`Key` and `dest_name` are encrypted with *A*'s private key. `Cert_string` is a random string which will be used to certify owners from *B* to *A* (see Section 7.2.5.2). `Encapsulation` indicates the type of security encapsulation to be used for future control messages and data message headers from *A* to *B*.[11] The entire request is then encrypted with *B*'s public key.

The reply message has the form:

```
msg_typedef struct {
        SKE_KEY                     key;            // the channel key
        BYTES                       dest_name;      // the name of the destination
        CERT_STRING                 cert_string;
        SECURITY_ENCAPSULATION      encapsulation;
        CHECKSUM                    checksum;
} ESTABLISH_SECURE_CHANNEL_REPLY;
```

`Key` and `dest_name` are encrypted with *B*'s private key. `Cert_string` is a random string which will be used to certify owners from *A* to *B* (see Section 7.2.5.2). `Encapsulation` indicates the type of security encapsulation to be used for future control messages from *B* to *A*. The entire reply message is encrypted with *A*'s public key.

In both the request and reply messages, the presence of the destination name encrypted with the sender's private key provides authentication, while the encryption of the entire message with the receiver's public key provides privacy. This use of slow public key encryption allows the ST to "bootstrap" into the faster single-key encryption using the connection key.

7.2.5.1. Control Message Structure

For control messages other than secure connection establishment, the security encapsulation depends on the parameters of the underlying network channel. The bit error rate for control messages must satisfy an upper bound determined by the two hosts. In addition, control messages are host-authenticated. They may also be private; this is a parameter of the host.

[11] There could also be a negotiation between *A* and *B* to determine the encapsulations of control messages and data message headers, perhaps independently. This could also be done on a per-network-channel basis.

The following header is common to all ST control protocol messages:

```
msg_typedef enum {
      CERTIFY_ASK
      CERTIFY_OFFER,
      CERTIFY_ASK_AGAIN
      CREATE_CHANNEL,
      DELETE_CHANNEL,
      FAST_ACK,
      PING
} ST_CONTROL_OP;

msg_typedef struct {
      flags {request};                          // true iff request; else reply
      ST_CONTROL_OP     operation;
      U32               my_seqno;
      U32               your_seqno;
} ST_CONTROL_HDR;
```

My_seqno is the sequence number of the next operation to be generated by the sender (or the current operation, in the case of a request message). Your_seqno is the sequence number of the next operation expected from the peer (or the operation being replied to). This header is followed by a message body whose structure depends on the message type.

The ST control protocol uses a simple retransmission policy. A request message is periodically retransmitted until a reply is received. Duplicate request messages are ignored. There are no reply acknowledgements. A reply message is retransmitted only if a message is received with a sequence number indicating that the reply was lost.

7.2.5.2. Owner Certification

An owner *O* is said to be *certified from host A to host B* if *B* believes that *A* possesses *O*'s private key. An owner *O* becomes certified from *A* to *B* if either

(1) *B* trusts *A*, and *A* informs *B* that it has *O*'s private key, or

(2) *A* has proved to *B* that it possesses *O*'s private key. To do this, *A* encrypts the pair <*R*,*B*> with the *O*'s private key, where *R* is the certification string provided by *B* on secure connection establishment.

7.2.5.3. Certification Caching

When an owner *O* has been certified from *A* to *B*, both *A* and *B* will each have an OWNER object for *O*, and each will have issued a remote reference to the other for their version of this object. *A*'s remote reference table entry for this object has flags indicating whether *O* has been certified 1) from *A* to *B*, and 2) from *B* to *A*. These entries serve as an *owner certification cache*. Subsequent operations requiring that *O* be certified from *A* to *B* can simply consult this cache.

Each remote reference entry also contains an *invalid time* beyond which the certification from the peer (if any) is no longer valid (this value is obtained from the name service entry for the owner). If a reference is made beyond this time, then the owner must be recertified. If the user public key has not changed or the owners are mutually trustful, then recertification is not done.

7.2.5.4. Certification Messages

The CERTIFY_OFFER operation is used to offer an (unsolicited) certificate to another host.

```
msg_typedef struct {
      ST_CONTROL_HDR     header;
      REMOTE_REF         owner_ref;
      BYTES              owner_name;
      CERTIFICATE        certificate;
} CERTIFY_OFFER_REQ;

msg_typedef struct {
      ST_CONTROL_HDR     header;
      flags              {accepted};
      REMOTE_REF         ref;
} CERTIFY_OFFER_REPLY;
```

If the sender already has a remote reference to the owner, it is passed in owner_ref and owner_name is empty; otherwise owner_reef is NULL_REF and the owner name is passed explicitly, in which case a remote reference is returned in the reply. Certificate is the certification string, encrypted with the owner's private key, or empty if the peer trusts this host.

The CERTIFY_ASK operation is used to request that the peer authenticate an owner.

```
msg_typedef struct {
      ST_CONTROL_HDR     header;
      REMOTE_REF         owner_ref;         // owner remote reference (if known)
      BYTES              owner_name;        // symbolic owner name
} CERTIFY_ASK_REQ;

msg_typedef struct {
      ST_CONTROL_HDR           header;
      enum {OK, FAILED}        status;
      REMOTE_REF               owner_ref;
      CERTIFICATE              certificate;
} CERTIFY_ASK_REPLY;
```

The CERTIFY_ASK_AGAIN operation is used to request re-certification of an owner.

```
msg_typedef struct {
      ST_CONTROL_HDR     header;
      REMOTE_REF         owner_ref;
} CERTIFY_ASK_AGAIN_REQUEST;

msg_typedef struct {
      ST_CONTROL_HDR           header;
      enum {OK, FAILED}        status;
      CERTIFICATE              certificate;
} CERTIFY_ASK_AGAIN_REPLY;
```

7.2.5.5. ST Channel Creation

Each ST channel is multiplexed onto an existing network channel. An ST module can create an ST channel only on a network channel that it owns.

A channel creation request supplies the following parameters for each channel being requested:

```
msg_typedef struct {
        CHANNEL_DIRECTION direction;         // CHANNEL_OUT or CHANNEL_IN
        CHANNEL_PARAMS    params;
        REMOTE_REF        net_channel_ref;
        REMOTE_REF        st_channel_ref;
        U32               opaque_data;       // passed from active to passive
} ST_CONTROL_CHANNEL_DESC;
```

Net_channel_ref indicates which data network channel is to be used for the ST channel; it must be a channel owned by the sender. St_channel_ref is a reference to the local channel endpoint object.

The create request message has the following structure:

```
msg_typedef struct {
        ST_CONTROL_HDR             header;
        U32                        opaque_data;
        ENCAPSULATION_TYPE         encapsulation_type;
        REMOTE_REF                 notification_mpo;
        U32                        number;
} ST_CONTROL_CREATION_REQUEST;
```

This is followed by a sequence of ST_CONTROL_CHANNEL_DESC fields. Opaque_data will be included in the notification message delivered on the passive side to the notification_mpo.

The reply message has the following structure:

```
msg_typedef struct {
        ST_CONTROL_HDR             header;
        enum {ACCEPT, REJECT}      status;
        U32                        opaque_data;      // from passive to active
} ST_CONTROL_CREATION_REPLY;
```

If the request was accepted, this is followed by a sequence of REMOTE_REF and U32 fields, referring to the ST endpoint objects at the passive end and the opaque data passed from the passive to active client for each channel.

7.2.5.6. ST Channel Deletion

The DELETE_CHANNEL operation uses the following messages:

```
msg_typedef struct {
        ST_CONTROL_HDR             header;
        U32                        number;
} DELETE_CHANNEL_REQUEST;

msg_typedef struct {
        ST_CONTROL_HDR             header;
} DELETE_CHANNEL_REPLY;
```

The request message is followed by a list of remote references to the ST channels to be deleted.

7.2.5.7. Fast Acknowledgements

A fast acknowledgement message has the following structure:

```
msg_typedef struct {
        ST_CONTROL_HDR      header;
        REMOTE_REF          st_channel_ref;
        U32                 opaque_data;
} FAST_ACK;
```

Such a message acknowledges the receipt of a client message with the given opaque data on the ST channel identified by st_channel_ref.

7.2.6. The ST Data Protocol

The ST data protocol uses a set of network channels disjoint from the control connection channels. There are two types of ST data messages:

- A *simple* message: used to send one ST client message.
- A *fragment* message: used to send a fragment of an ST client message. This is needed when the client message (with headers) exceeds the network channel maximum message size.

An ST data message consist of an *ST header* followed by user data. The security encapsulation is determined as follows: the ST header must have a bit error rate below a system-defined value, and must be authenticated. The user data must satisfy the ST channel parameters. The ST header of a simple data message has the following structure:

```
msg_typedef struct {
        ST_DATA_TYPE        type;               // SIMPLE in this case
        flags {fast_ack,  auth_sender}
                            message_flags;
        U32                 st_seq_num;         // a unique ID for this message
        U32                 ack_id;             // for fast acks
        REMOTE_REF          dest_st_channel;
        REMOTE_REF          dest_mpo;
        REMOTE_REF          sender;
        U32                 data_size;
} ST_DATA_SIMPLE;
```

Dest_st_channel refers to the receiving ST channel endpoint object. Dest_mpo refers to the MPO to which the message is to be delivered. If it is NULL_REF, the message is delivered to the default MPO. The optional sender refers to the OWNER object responsible for sending the message.

The header of a fragment message has the following structure:

```
msg_typedef struct {
        ST_DATA_TYPE        type;               // FRAGMENT in this case
        U32                 total_frags;        // number of fragments in message
        U32                 frag_num;           // number of this fragment
        U32                 st_seq_num;         // a unique ID for this message
        U32                 ack_id;             // for fast acks
        REMOTE_REF          dest_st_channel;
        REMOTE_REF          dest_mpo;
        REMOTE_REF          sender;
        U32                 data_size;
} ST_DATA_FRAG;
```

8. THE REMOTE OPERATION FACILITY

The DASH *Remote Operation Facility* (ROF) provides host-to-host request/reply communication. It supports higher-level request/reply communication (such as service access by user processes), as well as direct kernel communication. The following are the most important features of ROF:

- The channel approach is used: low-delay channels are used for critical-path messages such as requests and replies, and high-delay channels are used for other messages such as retransmissions and acknowledgements.
- There is no restriction on the number of outstanding remote operations between two DASH hosts. This removes a possible limit on the parallelism within a multiprocessor host.
- ROF does not dictate any mechanism for accessing messages (e.g., serialization/deserialization). Clients and servers access messages directly using DML, the DASH Message Language (see Appendix I).

8.1. Reliability

ROF provides three semantics for remote operations:

Exactly Once

In the absence of machine or network failures, operations are executed exactly once, and in any case are not executed more than once. Operations may have reply messages.

At Least Once

In the absence of machine or network failures, operations are executed at least once, and possibly more than once. Operations may have reply messages. This operation type can be used for idempotent operations.

Maybe

Operations are executed 0 or 1 times; the client is not told which. There can be no reply message. This operation type can used to distribute hints or other non-critical information.

8.2. Remote Operation Opcodes

A remote operation (RO) is identified by a 32-bit *RO opcode*. The set of RO opcodes is divided as follows:

- *Mandatory* operations that must be supported on all DASH hosts.
- *Optional* operations whose semantics are globally defined, but that need not be implemented on all hosts.
- *Non-standard* operations, which are specific to a particular kernel type.

8.3. Invoking a Remote Operation

ROF is represented by an ROF module, which provides the following interface for invoking an RO:

```
RO_STATUS
ROF::ro_call (
      HOST*               destination,
      U32                 ro_opcode,          // RO opcode - discussed above
      MESSAGE*            request,            // request message
      MESSAGE**           reply,              // reply message (optional)
      RO_TYPE             ro_type             // EXACTLY_ONCE, AT_LEAST_ONCE, MAYBE
);

enum RO_STATUS {
      OK,
      INVALID_OPCODE,
      OPCODE_NOT_SUPPORTED,
      NO_CONNECTION,              // unable to establish a ROF connection
      ERROR_OTHER,
};
```

If the reply is NULL and the status is OK, there was no reply. ROF clients may use the AUTHENTICATE_SENDER and AUTHENTICATE_RECEIVER flags in request and reply messages (See section 7.1.5).

8.4. ROF Server Interface

A REQ_REPLY_MPO object is associated with each RO opcode using the following operation:

```
void
ROF::register_ro_mpo (
      U32                 ro_opcode,
      REQ_REPLY_MPO*      ro_mpo
);
```

When a request is received, the ROF module looks up the RO opcode. If it is invalid or is an operation the server does not support, ROF sends a reply message with the appropriate error code. Otherwise, a request_reply() operation is performed on the associated MPO.

8.5. ROF Connection Parameters

Communication between peer ROF modules uses a dedicated set of ST channels (see Section 8.6). ROF does not support fragmentation of request or reply messages. The maximum request and reply message size is determined by the maximum message size of the ST channels that ROF is using. The following functions return these sizes:

```
int
ROF::max_request_message (
      HOST* remote_host         // destination host
);

int
ROF::max_reply_message (
      HOST* remote_host         // destination host
);
```

ROF client and servers may have differing security (authentication and privacy) needs. For simplicity, ROF uses private and host-authenticated channels.

8.6. The ROF Protocol

8.6.1. ROF Connections

All ROF communication between a particular client/server host pair uses a dedicated set of ST channels called the *ROF connection*. The connection is created by the client ROF module. A ROF connection is directional; it is *outgoing* relative to the client, *incoming* relative to the server. If two ROF modules are each acting as a server for the other, then there must be two ROF connections between them.

Separate fast and slow channels are used in each direction in order to reflect the relative deadlines of messages. A ROF connection consists of four channels: the `FAST_REQUEST_CHANNEL` and `SLOW_REQUEST_CHANNEL` go from the client to the server, and the `FAST_REPLY_CHANNEL` and `SLOW_REPLY_CHANNEL` go from the server to the client. Initial request and reply messages always use the fast channels, while retransmissions and acknowledgements use the slow channels.

The channels in a ROF connection are ''logical'' in that

- They may not be distinct: the fast channel and the slow channel in a given direction may actually be the same channel.
- There may be more than one actual channel for each logical channel; this might be necessary if the capacity of one ST channel is insufficient.
- The channels may change over the life of a ROF connection; if one of them is closed (e.g., because of network failure), the ROF module may create a new channel without breaking the ROF connection.

The client ROF module creates the ROF connection to a peer host when it has a request for that host and a ROF connection does not already exist. This occurs, for example, after one of the hosts comes up from a crash. ST notification messages are sent to the *ROF notification port*, which has a well-known remote reference (see Section 5).

ST allows the creator of an ST channel to pass opaque data in the notification message. ROF uses this to allow the server to distinguish between the different channels of the ROF connection; there is a different code for each of the four channels. In addition, a bit is used to specify whether or not this is a new connection. This is necessary to support reestablishment of individual channels of the ROF connection.

The ROF client owns the channels in the ROF connection, and it may delete the connection. It does so by using `SUBTRANSPORT::delete_channel()` to delete the channels. No messages need to be passed between the ROF modules. The server may request to delete the ROF connection. This may be necessary if the server is going down.

8.6.2. ROF Messages

A remote operation (RO) may involve a request message, a reply message, and various retransmissions and acknowledgement messages. An *ROF transaction* is the set of all messages associated with a single RO.

ROF messages are of two types: *ROF client messages* sent by the ROF client, and *ROF server messages* sent by the ROF server. A ROF message consists of two parts: a header containing control information, optionally followed by data. The two parts are sent together as a single message on an ST channel.

There are two header formats, one for client messages and one for server messages. To simplify message handling, each header type is fixed-size, regardless of the actual fields used.

8.6.2.1. Client Message Header Format

Client messages headers have the following format:

```
msg_typedef flags {
        EXACTLY_ONCE_REQUEST,
        AT_LEAST_ONCE_REQUEST,
        MAYBE_REQUEST,
        EXP_ACK_REQUESTED,
        PING_REQUEST,
        REPLY_ACK,
        DELETE_ACK
} CLIENT_FLAGS;

msg_typedef struct
        CLIENT_FLAGS        client_flags;        // message type
        U32                 ro_opcode;           // which operation
        U32                 seqno;               // ROF transaction ID
        REMOTE_REF          client_mpo;          // where to send reply
} ROF_CLIENT_HEADER;
```

Sequence numbers are generated by the client to identify each ROF transaction. The first transaction on a ROF connection may have any sequence number. Subsequent transactions must have strictly increasing sequence numbers. Sequence numbers may not repeat. `Client_mpo` is the remote reference to the MPO where the reply message is to be delivered to the client.

8.6.2.2. Server Message Header Format

Server message headers have the following format:

```
msg_typedef flags {
        REPLY,
        REQUEST_ACK,
        DELETE_REQ
} SERVER_FLAGS;

msg_typedef enum {
        SUCCESSFUL,
        INVALID_RO_OP,
        RO_OP_NOT_SUPPORTED,
} RO_STATUS;

msg_typedef struct {
        SERVER_FLAGS        server_flags;
        RO_STATUS           status;
        U32                 seqno;
} ROF_SERVER_HEADER;
```

8.6.3. ROF Protocol Specifications

The ROF exactly-once protocol uses implied request acknowledgements; that is, a reply acknowledges the corresponding request. Other systems such as Sprite [23] and Cedar

[5] also use implied reply acknowledgements (i.e., a request message acknowledges the previous reply). Implied reply acknowledgements are not used in ROF.

Tables 7.1 and 7.2 specify the ROF *exactly once* protocol. Tables 7.3. and 7.4 specify the ROF *at least once* protocol. It is assumed, for simplicity, that the ROF connection has already been established, and that there are no machine or network failures.

9. THE SERVICE ACCESS MECHANISM

Services are a class of remotely-accessible logical resources in the DASH distributed system architecture. The DASH *service access mechanism* (SAM) allows clients to name services and communicate with them in a uniform way. The goals of SAM are:

- To provide *replication transparency*. A service may consist of multiple *instances* running on different hosts. A client need not know which instance handles a particular request or session.
- To provide *location transparency* in the sense that service names do not specify or limit the location of the servers.
- To provide *failure transparency*: If a service instance fails, SAM may, without client involvement, locate and begin using a second instance of the service.
- To provide a flexible framework for client/service communication protocols. Services may provide interfaces that have real-time communication performance requirements, or that use special-purpose stream protocols.

Current State	Event		
	Receive Reply	Receive ACK	Timeout
STATE 1 Initial Request Sent. No explicit ACK requested.	Send Reply ACK. Goto State 4.	NA.	Send Duplicate Request. Ask For Explicit ACK. Goto State 2.
STATE 2 Duplicate Request Sent. Explicit ACK Requested.	Send Reply ACK. Goto State 4.	Goto State 3.	Send Duplicate Request. Explicit ACK Requested Goto State 2.
STATE 3 Request Sent. ACK of Request Received.	Send Reply ACK. Goto State 4.	Goto State 3.	Send Ping Message. Goto State 3.
STATE 4 Received Reply.	Send Reply ACK. Goto State 4	NA.	NA.

Table 7.1: Client State Table, ROF Exactly-Once Protocol.

Current State	Event				
	Receive Request	Receive Request ACK Requested	Receive Reply ACK	Receive PING Request	Finish Executing RO
STATE 1 Idle.	Execute RO. Goto State 2.	Send Request ACK. Execute RO. Goto State 2.	NA.	NA.	NA.
STATE 2 Executing RO.	Send Request ACK. Goto State 2.	Send Request ACK. Goto State 2.	NA.	Send Request ACK. Goto State 2.	Send Reply. Goto State 3.
STATE3 Reply Sent. No ACK Received.	Send Reply. Goto State 3.	Send Reply. Goto State 3.	Goto State 4.	Send Reply. Goto State 3.	NA.
STATE 4 RO transaction Complete.	NA.	NA.	NA.	NA.	NA.

Table 7.2: Server State Table, ROF Exactly-Once Protocol.

Current State	Event		
	Receive Reply	Receive ACK	Timeout
STATE 1 Initial Request Sent. No explicit ACK requested.	Send Reply ACK. Goto State 4.	NA.	Send Duplicate Request. Ask For Explicit ACK. Goto State 2.
STATE 2 Duplicate Request Sent. Explicit ACK Requested.	Send Reply ACK. Goto State 4.	Goto State 3.	Send Duplicate Request. Explicit ACK Requested Goto State 2.
STATE 3 Request Sent. ACK of Request Received.	Send Reply ACK. Goto State 4.	Goto State 3.	Send Ping Message. Goto State 3.
STATE 4 Received Reply. RO Transaction Complete.	NA.	NA.	NA.

Table 7.3: Client State Table, ROF At-Least-Once Protocol.

Current State	Event			
	Receive Request	Receive Request ACK Requested	Receive PING Request	Finish Executing RO
STATE 1 Idle.	Execute RO. Goto State 2.	Send Request ACK. Execute RO. Goto State 2.	NA.	NA.
STATE 2 Executing RO.	Send Request ACK. Goto State 2.	Send Request ACK. Goto State 2.	Send Request ACK. Goto State 2.	Send Reply. Goto State 1.

Table 7.4: Server State Table, ROF At-Least-Once Protocol.

9.1. The Service Abstraction

A DASH *service* is a set of *instances* that together form a logical resource. Each instance resides on a single host. An instance may consist of a process, a set of processes, or a ''registration'' with the host kernel that causes a process to be created as needed. Information about the services on a host are kept in stable storage (e.g., a configuration file) so that they survive crashes.

A replicated service may provide an abstraction of consistent data, in which case it needs to ensure consistency between its instances. DASH does not supply nor dictate any method for this, or for ensuring the atomicity or permanence of operations on services. Such mechanisms must be supplied by the services themselves, perhaps in cooperation with a higher-level transaction manager.

A DASH host may provide a *kernel service*, allowing access to resources that are inherently local to that host, such as its physical devices.

Services can be accessed in two basic modes:

Request/Reply mode

Operations are request/reply. Operations are conveyed to remote service instances via ROF.

Session mode

SAM uses ROF to contact an instance of the service and set up a communication channel between the client and server.[12] This allows clients and servers to communicate using specialized, dynamically configurable protocols.

9.1.1. Service Tokens

A service may issue a *service token* representing a name or object within the service. The token can thereafter be supplied in lieu of a name in subsequent operations on the service. A service token may be used only in accessing the service instance that issued

[12] The design for session mode access is not complete.

the token.

A service token has an associated set of operations, specified (by a bitmask) when the token is requested; the token provides the right to perform this set of operations on the object to which it refers, bypassing any underlying protection mechanism. A token ma;· have no access rights, in which case it serves only as a name abbreviation.

The use of service tokens can improve performance in two ways: 1) it eliminates the need for the service to do per-operation authorization checking; 2) it eliminates the need for the service to do per-operation name translation. The DASH service token mechanism is related to the V system's UIO interface [9].

Service tokens can serve as capabilities (albeit transient ones), and therefore must be protected. There are two possible approaches:

- The token includes a large random part, is secret, and must be encrypted in network messages.
- The token need not be encrypted in network messages, and may be a small integer. The service accepts a token only from its original recipient.

DASH uses the second approach. A service maintains, in per-host tables, the set of service tokens it has issued.

Service tokens may be discarded at any time by a service. This may be done either to limit table size, or to force periodic reauthorization in support of an ''eventual revocation'' policy. The client (or the client kernel) holding the token must store information (name and operation set) used to obtain the token, and must be prepared to issue another token request if the original token is invalidated by the service. In addition, the client can ''release'' the token, providing a hint to the service that it discard the token.

A token does not have associated with it any session context (e.g., position within a file). Two tokens representing the same name and having the same rights are interchangeable.

Tokens are usable only during a crash-free period on both the client and the service instance. When a host loses a secure channel to a remote host, all tokens associated with services running on the remote host are discarded.

9.2. Client Interface to SAM

SAM is represented by a SAM module. SAM provides the following function to perform an operation:

```
SAM_STATUS
SAM::operation (
        NAMED_ENTITY*       prefix,      // NAME_SERVICE, SERVICE, SERVICE_TOKEN
        char*               extension,   // relative to prefix
        OWNER*              owner,
        MESSAGE*            request,
        MESSAGE**           reply
);
```

```
enum SAM_STATUS {
        OK,                           // operation was successful
        SERVICE_FAILURE,              // service was unavailable
        INVALID_PREFIX,               // prefix was invalid
        INVALID_EXTENSION,            // syntactically invalid extension
        NO_SUCH_EXTENSION,            // could not resolve name
        NO_AUTHORIZATION,             // authorization failure
};
```

Names are specified using `prefix` and `extension`. `Prefix` may point to a `NAMED_ENTITY` object or be `NULL`. `Extension` is relative to `prefix`. If `prefix` is `NULL`, then `extension` is a complete name. Because services may do authorization based on the owner name, the owner requesting the operation is specified in `owner`.

Service tokens are represented in the client kernel by objects of class `SERVICE_TOKEN` (derived from `NAMED_ENTITY`). `SAM` provides the following operation for creating these objects:

```
SAM_STATUS
SAM::get_service_token(
        NAMED_ENTITY*       prefix,      // NAME_SERVICE, SERVICE, SERVICE_TOKEN
        char*               extension,   // relative to prefix
        OWNER*              owner,       // used for authorization
        char*               operations,  // operations associated with token
        SERVICE_TOKEN**     token        // new token
);
```

The meaning and format of `operations` is service-specific. The new service token is returned in `token`.

9.3. Server Interface to SAM

Each instance of a service is identified by a pair consisting of

- The name of the host on which it runs.
- An *instance ID*, a 32-bit ID unique among service instances on that host.

Services have symbolic names in the DASH global name space (see Section 10). Each service name is mapped (by the DASH name service) to a list of (*host name*, *instance ID*) pairs. A service may have several different names. Only those instances of a service that are intended for remote access need be listed in the name service entry. For example, a file service may have *local instances* on work stations. These local instances might do local caching, and never be accessed remotely.

The steps in offering a service that will run as a user process on the DASH kernel are as follows:

(1) Write a program that implements the *local control protocol* (see Section 9.4).

(2) Make versions of this program for the hosts on which instances are to be run.

(3) Register the service with the `SAM` module on each of these hosts (see below).

(4) Register the service with the DASH name service.

Services can be registered and unregistered using:

```
SAM::register(
      REQ_REPLY_MPO*      service_mpo,
      SVC_ID              instance_id
);

SAM::unregister(
      SVC_ID        instance_id
);
```

`Service_mpo` is a request/reply MPO to which requests to the service will be delivered. Depending on the nature of the service, this object may:

- Perform a function call within the kernel.
- Deliver the message to a user-level server process, perhaps creating a VAS and a process if necessary.

The `SAM` module maintains a table of service registrations, mapping service ID's to request/reply MPO's.

9.4. SAM Protocols

There are three protocols involved in service access:

(1) A protocol between peer SAM modules, layered on top of ROF. This is called the *SAM network control protocol.*

(2) A protocol between a server-side SAM module and a local service instance. This is called the *SAM local control protocol.*

(3) The service-specific end-to-end protocol between a client and a service, defining the format and semantics of the service's operations. This is called the *SAM client/service protocol.*

9.4.1. Network Control Protocol

The SAM network control protocol consists of remote operations using the ROF facility. In the DASH kernel, these operations are generated and handled by SAM modules. In specialized server machines, they might be handled by the service itself. The SAM network protocol uses the following set of ROF opcodes:

```
SVC_TOKEN_REQUEST
SVC_OPERATION
```

The `SVC_OPERATION` operation is used to perform an operation on a service. The request message has the following structure:

```
msg_typedef struct {
      U32         service_id;
      U32         service_token;
      BYTES       extension;
      BYTES       request;
} SVC_OPERATION_REQUEST;
```

`Service_token` is a service token previously issued by the service, or `NULL`. `Extension` is a name extension beyond that of the token, or beyond the name of the service if no token is used.

The reply message is either an operation result from the service, an error return, or a REDIRECT message forwarding this request to another service instance. In this case the message contains a *(host name, instance ID)* pair, which is a hint for which instance to try next.

The SVC_TOKEN_REQUEST operation is used to obtain a service token. It uses the following messages:

```
msg_typedef struct {
        U32          service_id;
        U32          old_token;
        BYTES        extension;
} SVC_TOKEN_REQUEST;

msg_typedef struct {
        TOKEN_REPLY_STATUS       status;
        U32                      new_token;
} SVC_TOKEN_REPLY;
```

In the request message, old_token is an optional service token to which extension is relative.

9.4.2. Local Control Protocol

The protocol between a server-side SAM module and a service instance uses the same request/reply messages as the SAM protocol, except:

- Messages are in host byte order instead of network byte order.
- Some error codes (e.g., NO_SUCH_ID) are not used.

10. GLOBAL NAMING

A primary goal of the DASH communication architecture is that resources (data and computational) should be uniformly and securely accessible from any host. This requires a facility for naming and locating the resources, and for naming and authenticating resource owners and clients. These functions are provided by the *DASH global naming system.*

10.1. Name Space Structure

The DASH global naming system uses a single tree-structured name space, similar to that described in [6]. Conceptually, a name is a list of *components*, each of which is a string of printable ASCII characters not containing the character ''/''. In practice, a name is represented as a single ASCII character string consisting of a sequence of components separated and preceded by ''/''. For example,

```
/usa/uc-berkeley/computer-science/filer
```

is a name with four components. A ''/'' by itself constitutes a name with zero components; this is the name of the *root* of the naming tree.

10.2. Entity Types

The system is used to name four types of entities: *hosts*, *owners*, *services* and *name services*. The internal nodes of the tree represent name services, and the leaves of the tree represent the other entity types. The entity types, and their associated attributes, are as follows:

- An **owner** represents an individual human user or a "role" such as that of system manager. Its attributes include its public key.
- A **host** is an endpoint of physical network communication. Its attributes include a list of its network addresses, and the name of its owner.
- A **service** is a logical resource provided by set of programs or processes. Its attributes include 1) a list of *(host name, instance ID)* pairs each specifying an instance of the service (see Section 9.1) and 2) the name of the owner of the service. Services that are not name services (see below) are called *general services*.
- A **name service** is a special type of service that manages a single *directory*. Each entry in a directory has a name component, a type, and a set of attributes. The attributes of a name service include those of general services (i.e., the service owner and the set of hosts on which instances exist), but also include the attributes of the hosts, the attributes of the host owners, and the attributes of the owner of the name service. This extra information is included to avoid infinite recursion in the name resolution process (see Section 10.5.3).

In addition to the *mandatory* attributes listed above, each name service entry may have an arbitrary-length character string for *auxiliary* information. This typically would be structured as a set of *name=value* string pairs. It could be used to store attributes such as the real-life name, US mail address, electronic mail server address, phone number of an owner, or the access protocol used by a service.

Each name service entry also has a "cache time" field indicating the maximum amount of time for which it should be cached by clients (kernels or other name services). There is no cache consistency protocol, so resolutions may be incorrect during the cached period. Any intermediate agent (e.g., another name server or a kernel) that caches name entries should maintain the amount of time for which it has held each entry and invalidate the entry when the cache time expires. If a name server releases the entry to another agent, it should replace the cache time field with a suitably reduced value.

10.3. Intra-Service Naming

SAM allows services other than name services to extend the global name space below their own name. Hence they can provide global names for the objects they manage. Such a name has the form

```
service-name/intra-service-name
```

where *service-name* is the name of a general service. For example, a file service might provide hierarchical naming of its files, so that the name

```
/usa/uc-berkeley/cs/filer/anderson/foo
```

refers to the file

```
/anderson/foo
```

within the file service

```
/usa/uc-berkeley/cs/filer .
```

This feature removes the need to distinguish the two levels of naming, and makes it possible for services to provide named "sub-services". In addition, services can provide non-hierarchical intra-service naming. For example, a file service supporting attribute-based naming might provide a name of the form

```
/usa/uc-berkeley/cs/filer2/anderson.dash.kernel.c-source.scheduler
```

specifying a file (or group of files) with attributes *anderson*, *dash*, *kernel*, *c-source*, and *scheduler*.

10.4. The Interface to Naming in the DASH Kernel

In the DASH kernel, the interface to the global naming system is provided by the NAMING module. The basic function of this module is name resolution. That is, given a global name *N*, it finds the longest prefix of *N* that names a standard entity (owner, host, general service, or name service), and returns 1) the attributes of that entity, and 2) the remainder of *N* beyond the name of the standard entity. The interface for resolving a name is:

```
NAMING_STATUS
NAMING::resolve (
        NAME_SERVICE*      prefix,            // a NAME_SERVICE or NULL
        char*              extension,         // relative to prefix
        OWNER*             owner,             // used for authorization
        NAMED_ENTITY**     result_prefix,     // standard entity (returned)
        char**             result_extension   // name remainder, could be NULL
);


enum NAMING_STATUS {
        OK,                         // resolution was successful
        NO_SUCH_NAME,               // no such name is known
        INVALID_NAME,               // syntactically invalid name
        NO_AUTHORIZATION,           // name service authorization failure
        NAME_SERVICE_FAILURE,       // a name service was unavailable
};
```

Names are specified using a `prefix` and an `extension`. `Prefix`, if non-NULL, points to an object (returned by a previous call to `NAMING::resolve()`) representing a name service. `Extension` extends the name represented by `prefix`. If `prefix` is `NULL`, `extension` is a global name. `Owner` specifies the owner on whose behalf the name is being resolved. This is relevant if authorization is used by any of the name services involved. If the resolution is successful, the results are placed in `result_prefix` and `result_extension`. Resolution fails if 1) the name is syntactically invalid, 2) a directory lookup fails, 3) an authorization failure occurs, or 4) a name service fails.

10.5. The Name Service Protocol

This section specifies the minimal set of operations that must be supported by a DASH name service. These operations may be invoked by other name services acting on behalf of their clients, or by the clients themselves. A name service may provide other operations as well, such as those needed for administration or authorization changes.

10.5.1. Name Resolution

Every name service may be asked to resolve any name. In addition to various error returns, the following results are possible:

(1) If this resolution is successful, the attributes of the standard entity with the longest prefix are returned.

(2) If the queried name service is unable to authenticate the requesting owner to a protected intermediate name service (i.e., because it does not have the owner's private key), the entry for the intermediate name service is returned. The client's kernel must then contact this name service directly.

The name resolution messages are [13] :

```
msg_typedef struct {
      SERVICE_TOKEN      prefix;
      BYTES              extension;
      RESOLVE_REQ_FLAGS flags;           // see below
} NAMING_RESOLVE_REQUEST;

msg_typedef struct {
      U32                status;          // OK or AUTHORIZATION_ERROR
      U32                component_num;   // where resolution finished
      NS_ENTRY           entry;           // attributes of entry
} NAMING_RESOLVE_REPLY;

msg_typedef flags {
      no_cache                 // do not use cached information
} RESOLVE_REQ_FLAGS;
```

10.5.1.1. Name Tokens

Global names are potentially very long. Even with caching in name services, component-by-component resolution may yield unacceptable performance. To confront this problem, name services may supply service tokens (Section 9.1.1) representing names. This particular type of service token is called a *name token*. Name tokens have no associated access rights. The same token may be issued to any number of clients, but is valid only for the name service instance that issued it. Typically, a name service would use a name token as a reference into its cache.

Unlike other service tokens, a name token does not represent a name that extends that of the issuing name service. Rather, it represents a global name.

In the DASH kernel, the NAMING module uses the name token mechanism internally to speed name resolution. This is done transparently to the clients of NAMING.

10.5.2. Scan Operations

Scan operations are used to read the set of entries in a name service's directory. This facility can be used to provide for resolution based on incomplete information, ''wildcard'' queries, and so on.

[13] NS_ENTRY is not specified. See Section 10.2 for a list of the attributes for each type of named entity.

Using flags passed in the request message, the operation can be limited to a subset of the entry types. Also using flags, the information returned for each entry can be limited to its type.

Since the number of entries in a directory may be large, the scan operation may return a subset of the entries. The operation can specify an initial entry number, and the client can make a sequence of scan operations to scan the entire directory.

The following messages are used to scan directories:

```
msg_typedef struct {
      U32   start_index;
      flags {
            services,           // return entries of type SERVICE
            name_services,      // return entries of type NAME_SERVICE
            hosts,              // return entries of type HOST
            owners,             // return entries of type OWNER
            return_types,       // return type of each entry
            return_names,       // return name of each entry
            return_attrs        // return complete attributes
      } flags;
} SCAN_REQUEST;

msg_typedef struct {
      SCAN_STATUS         status;             // see below
      U32                 num_entries;        // number of entries in reply.
      SCAN_ENTRY          entries[];          // array of information
} SCAN_REPLY;

msg_typedef struct {
      BYTES name;
      union {
            NS_ENTRY            ns_entry;     // complete attributes requested
            NS_ENTRY_TYPE       type;         // only entry type requested
            NULL;                             // only name was request
      };
} SCAN_ENTRY;

msg_typedef enum {
      OK,                 // scan was successful
      NO_AUTHORIZATION    // did not have correct authorization
} SCAN_STATUS;
```

10.5.3. Avoiding Infinite Loops in Name Resolution

The attributes of a general service include 1) a list of (*host name, instance ID*) pairs for each instance of the service, and 2) the name of the owner of the service. The attributes for a name service include those of a general service, but also include the attributes of the hosts where there are instances of the service, the attributes of these host's owners, and the attributes of the owner of the name service.

To see why it is necessary to include this extra information, consider the following scenario. Assume that the name service /edu/davis has only one instance, which runs on the host /edu/davis/host1. A client on the host whose parent name service is /edu tries to resolve the name /edu/davis/host1. The host asks /edu to resolve /edu/davis/host1. /edu has an entry for /edu/davis. If /edu returned only the service attributes for `davis` (e.g., a list of host names and instance ID's), the next step in the resolution process would be to resolve the name `/edu/davis/host1`, which the original request. The extra information outlined above solves this problem.

11. ACKNOWLEDGEMENTS

The following people have contributed to this work: Kevin Fall, Shin-Yuan Tzou, Raj Vaswani, and Giuseppe Facchetti.

REFERENCES

1. D. P. Anderson and P. V. Rangan, "A Basis for Secure Communication in Large Distributed Systems", *IEEE Symposium on Security and Privacy*, Apr. 1987.

2. D. P. Anderson, D. Ferrari and P. V. Rangan, "Subtransport Level: The Right Place for End-to-End Security Mechanisms", Technical Report No. UCB/Computer Science Dpt. 87/346, Computer Science Division, EECS, UCB, Berkeley, CA, Mar. 1987.

3. D. P. Anderson and D. Ferrari, "The DASH Project: An Overview", Technical Report No. UCB/Computer Science Dpt. 88/405, Computer Science Division, EECS, UCB, Berkeley, CA, Feb. 1988.

4. M. Bastian, "Voice-Data Integration: An Architecture Perspective", *IEEE Commun. Mag. 24*, 7 (July 1986), 8-12.

5. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.

6. A. D. Birrell, B. W. Lampson, R. M. Needham and M. D. Schroeder, "A Global Authentication Service without Global Trust", *IEEE Symposium on Security and Privacy*, 1986.

7. D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *Proc. of the 9th ACM Symp. on Operating System Prin.*, Bretton Woods, New Hampshire, Oct. 10-13, 1983, 128-140.

8. D. R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems", *1986 SIGCOMM Symposium*, Aug. 1986, 406-415.

9. D. R. Cheriton, "UIO: A Uniform I/O System Interface for Distributed Systems", *Trans. Computer Systems 5*, 1 (Feb. 1987), 12-46.

10. D. D. Clark, M. L. Lambert and L. Zhang, "NETBLT: A High Throughput Transport Protocol", *SIGCOMM87*, , 353-359.

11. J. Gettys, "Problems Implementing Window Systems in UNIX", *Proceedings of the 1986 Winter USENIX Conference*, Denver, Colorado, January 15-17, 1986, 89-97.

12. J. B. Jones and R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Distributed Object-Oriented Systems", *OOPSLA Conference Proceedings*, 1986.

13. B. Lyon, "Sun Remote Procedure Call Specification", *Sun Microsystems, Inc. Technical Report*, 1984.

14. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Comm. of the ACM 19*, 7 (July 1976), 395-404.

15. J. Nagle, "Congestion Control in IP/TCP Internetworks", *Internet RFC 896*, Jan. 1984.

16. ''Data Encryption Standard'', *FIPS Publication 46*, Washington, D.C., 1977.

17. J. Postel, ''Transmission Control Protocol'', *DARPA Internet RFC 793*, Sep. 1981.

18. J. Postel, ''Internet Control Message Protocol'', *DARPA Internet RFC 792*, Sep. 1981.

19. J. H. Saltzer, D. P. Reed and D. D. Clark, ''End-To-End Arguments in System Design'', *Trans. Computer Systems 2*, 4 (Nov. 1984), 277-288.

20. R. D. Sansom, D. P. Julin and R. F. Rashid, ''Extending a Capability Based System into a Network Environment'', *1986 SIGCOMM Symposium*, , 265-274.

21. B. Stroustrup, ''The C++ Programming Language'', *Addison-Wesley*, 1986.

22. D. B. Terry and D. C. Swinehart, ''Managing Stored Voice in the Etherphone System'', *Trans. Computer Systems 6*, 1 (Feb. 1988), 3-27.

23. B. Welch, ''The Sprite Remote Procedure Call System'', Technical Report No. UCB/Computer Science Dpt., Computer Science Division, EECS, UCB, Berkeley, CA, July 1986.

24. ''The DASH Local Kernel Structure'', UCB/Computer Science Dpt. Technical Report, in preparation, August 1988.

25. ''The DASH Virtual Memory System'', UCB/Computer Science Dpt. Technical Report, in preparation, August 1988.

APPENDIX I -- THE DASH MESSAGE LANGUAGE

1. Introduction

Local and remote interprocess communication in DASH is based on messages. The transport mechanism views a message as a logical array of bytes. For the sending and receiving clients, however, a message (or a portion of a message) may be interpreted as a collection of typed data items. A *structured message* is a (logically) contiguous portion of a DASH message that possesses a well-defined structure, or *type*.

This section describes the DASH facilities for structured messages. Specifically, it includes:

- The definition of a *DASH Message Language* (DML) for the specification of message structure. DML is not a general-purpose type definition facility, but rather is tailored to the limited needs of kernel-level clients.
- A *message representation standard* that dictates how messages with DML specifications are to be represented.
- A description of a preprocessor that converts a limited set of DML type definitions into a set of macros that facilitate building and accessing messages of those types.

This facility is related to components of RPC systems such as those of Mesa [5], Mach [12] and Sun UNIX [13]. The main difference between DML and these systems is that DML clients (message producers and consumers) are expected to access messages directly, rather than serialize and deserialize them. This is because the clients are DASH kernels, which demand efficiency rather than a transparent programming interface.

2. Message Type Definitions

DML is used to define named *message types*. The definition of a new type has the form

`msg_typedef` *type-definition type-name ;*

where *type-definition* is either the name of an existing type (such as the base types listed below) or one of the various *type constructors* defined in subsequent sections. A type name (or other user-assigned name in DML) may be any valid C identifier provided it does not contain two consecutive underscores and does not conflict with DML's keywords.

The following sections describe DML's base types and type constructors, and their associated representations as messages.

3. Base Types

The following set of *base types* is predefined:

```
enum {name1, name2, ...}
flags {name1, name2, ...}
U32, U64
S32, S64
BYTES
NULL
```

Enum is similar to the enum type in C. Flags denotes a set of up to 32 Boolean flags. U32 and U64 denote 4- and 8-byte unsigned integers, and S32 and S64 denote signed integers. BYTES denotes a variable-length array of bytes. NULL denotes an empty message; it is typically used as a union element.

The base types are represented as follows. The U and S types have length 4 or 8 bytes as appropriate. For network messages, the bytes are stored in network byte order (the highest order byte has the lowest address). The enum type is represented as a 32-bit unsigned integer, with values beginning from zero. The flags type is represented as a 4-byte word; bits are used from low-address to high-address byte, and from low to high-order bits within a byte. The BYTES type is represented as a 32-bit count and the data bytes, padded to a 4-byte boundary.

4. Structures

DML *structs* are a restricted form of structures (i.e., catenated labeled subtypes). The restrictions make it simpler to generate macros for building and accessing the structs. The facility is typically used to define the portions of messages (such as headers and trailers) that are accessed by a particular protocol level.

A struct is defined as follows:

```
struct {
        type1 label1;
        type2 label2;
        ...
}
```

Each *type* is either a base type or a previously defined struct type.

A struct is a mixed sequence of fixed-size fields and variable-length byte arrays. The representation (as a byte-array message) is as follows:

- If there are any BYTES fields, the first word of the message is the total length of the message.
- The fixed-size fields are placed together, in the order of their declaration, at the start of the message (after the total length word, if present).
- The fixed-size fields are followed by a vector of *(offset, length)* pairs for each of the BYTES fields, except for the first one (whose offset and length are implied by other information). The offset field is the distance in bytes from the start of the message to the beginning of the real bytes field; the length field is the length of the byte field (including possible padding).
- Each BYTES field is represented by the data bytes, padded (by unspecified bytes) to a 4-byte boundary.

Hence a struct with *n* fixed-size fields and *m* BYTES fields is represented as in Figure 1.

5. Preprocessor-Generated Macros

We will specify the output of the macro preprocessor by giving an example. Consider the following type definitions:

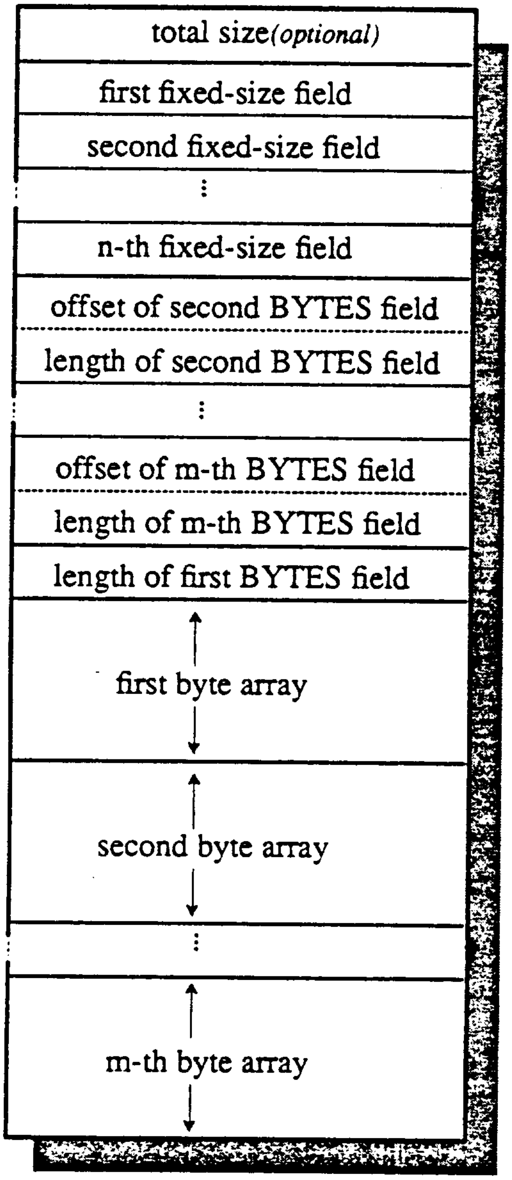

Figure 1: Internal Representation of a Structured Message.

```
expand REQ_MSG;

msg_typedef U32 SEQ_NO;

msg_typedef enum {OK, FAILURE} STATUS;

msg_typedef flags {SECURE, DEADLINE} MSG_FLAGS;
msg_typedef struct {
        SEQ_NO seq_no;
        MSG_FLAGS msg_flags;
        BYTES name;
} MSG_HEADER;

msg_typedef struct {
        MSG_HEADER header;
        enum {EXACTLY_ONCE, MAYBE} mode;
        BYTES info;
} REQ_MSG;
```

The expand declaration specifies the message types for which macros are to be generated (in this example, only REQ_MSG will be expanded).

The representation of a REQ_MSG message is as shown in Figure 2 (where *n* is the length of the name field, rounded up to a multiple of 4).

The DML preprocessor is given an input file, and a flag indicating whether the target machine uses network byte order, If it is run with the above definitions as input, it will generate the following macros.

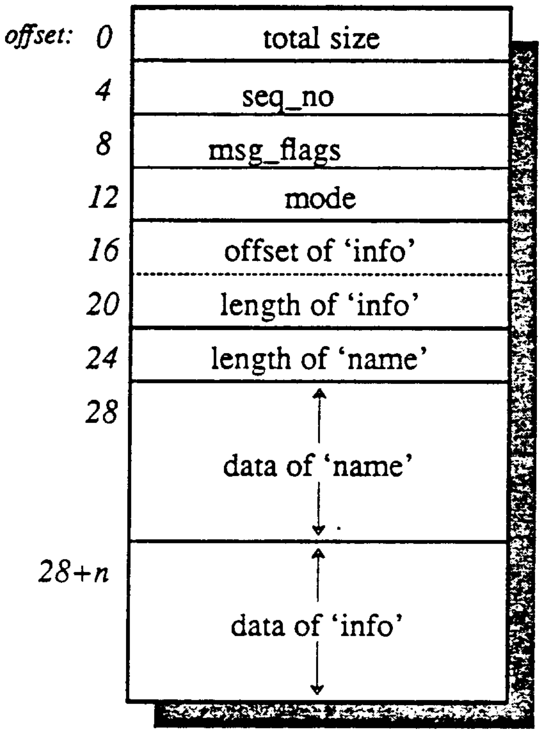

Figure 2: The Representation of a REQ_MSG Message.

5.1. Enum and Flag Definitions

The following macros define enum and flag values:

```
#define     STATUS__OK  0
#define     STATUS__FAILURE   1

#define     MSG_FLAGS__SECURE 1
#define     MSG_FLAGS__DEADLINE     2

#define     REQ_MSG__mode__EXACTLY_ONCE    0
#define     REQ_MSG__mode__MAYBE     1
```

5.2. Size Computation

Both the sender and the receiver of a message need to compute the total length of the message they are handling. This length depends on the size of the fixed-length fields (which can be calculated by the DML preprocessor) and on the variable length BYTES fields. If there are no variable-length fields, the total size of the message is a constant. If there are BYTES field, DML generates a macro

```
#define REQ_MSG__size(i, j)           (28 + mult4(i) + mult4(j))
```

used by the sender to determine the size of the message as a function of the lengths of the BYTES fields, and a macro

```
#define REQ_MSG__getsize(d)          *((U32 *) (d))
```

used by the receiver to get the total length stored in the message itself.

REQ_MSG__size computes the total size of a message instance, given values for the lengths of all BYTES fields. It uses a predefined macro

```
#define      mult4(i)     ((i+3)&0xfffffffc)
```

which computes the smallest multiple of 4 at least as large as its argument.

The argument of REQ_MSG__getsize is a pointer to the data part of a MESSAGE object. This function computes the size of the REQ_MSG instance occurring in the message. The message need not be contiguous, and is assumed to be in host byte order.

If there are no BYTES fields in the message, then the total length is a constant. In this case the two macros above expand to a constant value.

5.3. Initialization of Structural Information

This macro initializes the "bookkeeping" fields of a message (total length of the message and lengths and offsets of BYTES fields), given the lengths of its BYTES entries. If there are no BYTES entries, this macro is empty.

```
#define REQ_MSG__format(p, i, j) \
        *(U32 *) (p) = 28 + mult4(i) + mult4(j); \
        *(U32 *) ((p) + 24) = (i); \
        *(U32 *) ((p) + 16) = 28 + mult4(i); \
        *(U32 *) ((p) + 20) = (j);
```

5.4. Message Field Access

Given a contiguous message with bookkeeping data already in place, the following macros compute the address of the named field, and recast it to the proper type.

```
#define REQ_MSG__header__seq_no(p)     ((U32 *)   ((p) + 4))
#define REQ_MSG__header__msg_flags(p)  ((U32 *)   ((p) + 8))
#define REQ_MSG__header__name(p)       ((char *)  ((p) + 28))
#define REQ_MSG__header__mode(p)       ((U32 *)   ((p) + 12))
#define REQ_MSG__header__info(p)       ((char *)  ((p) + *((U32 *) ((p)+16))))
```

If REQ_MSG had contained structs having fixed size (that is, without any BYTES fields), then macros for accessing those structs would have been generated, too (with no recasting of pointers, though). This kind of macro is useful, for example, when the programmer needs to pass the address of a struct as a parameter in a function call.

5.5. BYTES Field Lengths

The following macros compute the actual length of each of the BYTES fields contained in the message.

```
#define REQ_MSG__header__name__length(p)      (* (U32 *) ((p) + 24))
#define REQ_MSG__header__info__length(p)      (* (U32 *) ((p) + 20))
```

These values, retrieved from the bookkeeping information of the message itself, show the actual space used by those BYTES fields, that is, they do not include the rounding to the next multiple of 4.

5.6. Byte Order Conversion

This macro converts a message instance to or from network byte order. Its argument is of type char*, and points to the message, which must be contiguous. If the target machine is big-endian (i.e., uses network byte order), this __byteorder macro is empty. On a little-endian machine, the macro is:

```
#define REQ_MSG__byteorder(p)    \
        BYTE_SWAP(p);           \
        BYTE_SWAP((p) + 4);     \
        BYTE_SWAP((p) + 8);     \
        BYTE_SWAP((p) + 12);    \
        BYTE_SWAP((p) + 16);    \
        BYTE_SWAP((p) + 20);    \
        BYTE_SWAP((p) + 24);
```

This macro is a sequence of calls to the predefined macro BYTE_SWAP, which converts a 32-bit word between host and network byte order (the same function goes both ways).

5.7. Example

The following operations must be performed in order to build a message using DML:

(1) Compute its length using __size.

(2) Initialize the bookkeeping fields using __format.

(3) Fill in the data fields.

(4) Immediately before sending a message to the network, and after receiving it from the network, use __byteorder().

6. Other Type Constructors

DML provides other type constructors. These are for documentation purposes only; they are checked for syntactic correctness, but no macros are generated for these types. They must appear at the end of a message definition.

6.1. List

A list, like a struct, represents a set of named subtypes. A list, however, places no restrictions on these subtypes. It is defined by:

```
list {
        type1 label2;
        type2 label2;
        ...
}
```

A list is represented by the catenation of its entries.

6.2. Union

A union (like a C union) represents a choice of one type out of a set of types. It is defined by

```
union {
        type1 label1;
        type2 label2;
        ...
}
```

The representation of a union is simply the representation of one of the subtypes. This differs from the C language, where the representation of a union is large enough to accommodate all subtypes.

6.3. Arrays

An array is a list of entries of a single (perhaps variable-size) type. It is defined by

```
type[const]
type[]
```

The first declares a fixed-length array, the second an array of indeterminate length. An array is represented by the catenation of its elements.